

XI
PROGRAMMER'S
GUIDE

♥ 1991-2011 Providence Software Solutions, Inc. All rights reserved.

The XI programming interface, XI manuals, and XI software may not be reproduced in any form or by any means except by permission in writing from Providence Software Solutions, Inc.

XI is a trademark of Providence Software Solutions, Inc.

XVT is a trademark of Providence Software Solutions, Inc.

Macintosh is a trademark of Apple Computers, Inc.

Microsoft Windows is a trademark of Microsoft, Inc.

Published by:

Providence Software Solutions, Inc.
202 New Edition Court
Cary NC, 27511
919-854-1800
919-4393034 (Fax)
<http://www.xvt.com> (Website)

Table of Contents

TABLE OF CONTENTS.....	III
TABLE OF FIGURES.....	VIII
1	
INTRODUCTION.....	1
1.1 WHAT IS XI?.....	1
1.2 LAYERS UPON LAYERS.....	2
1.2.1 XI.....	2
1.2.2 XVT.....	3
1.2.3 The Native System.....	3
1.3 CONSTRUCTING AN XI APPLICATION.....	3
1.3.1 XI Programming.....	4
1.3.2 XVT Programming.....	4
1.4 SUMMARY.....	5
2	
AN XI INTERFACE.....	6
2.1 SUMMARY OF XI OBJECTS.....	8
2.1.1 Objects.....	8
2.1.2 Events.....	9
2.1.3 XI Event Handler.....	11
2.2 SUMMARY.....	11
3	
CREATING AN OBJECT DEFINITION TREE.....	12
3.1 DESIGNING AN INTERFACE HIERARCHY.....	13
3.2 USING THE CONVENIENCE FUNCTIONS.....	13
4	
CHARACTERISTICS OF XI OBJECTS.....	16
4.1 COORDINATE SYSTEM.....	17
4.1.1 Form Units -vs- Pixels.....	18
4.2 CONTROL IDS.....	18
4.3 XI ATTRIBUTES.....	18
XIT_ITF4.4 INTERFACE OBJECTS.....	19
4.4.1 Modal Interfaces.....	19
4.4.2 Virtual Interfaces.....	19
4.4.3 Putting XI Interfaces in Existing Windows.....	21
4.4.4 Background Color.....	21

4.4.5 Menu Bars on Windows.....	22
4.4.6 Cutting and Pasting with XI.....	22
4.4.7 Scroll Bars.....	22
4.4.8 Close Box.....	23
4.4.9 Size Controls.....	23
4.4.10 Iconize Controls.....	23
4.4.11 Border Style.....	24
XIT_LIST4.5 LIST OBJECTS.....	24
4.5.1 Disabled lists.....	25
4.5.2 Enabled lists.....	25
4.5.3 No Column Headings.....	25
4.5.4 Horizontal Scrolling.....	25
4.5.5 Movable Columns.....	26
4.5.6 Resizing Columns.....	27
4.5.7 Dynamically Deleting Columns.....	27
4.5.8 Positioning and Inserting Columns in a List.....	28
4.5.9 List Button.....	28
4.5.10 Removing Horizontal and Vertical Rules of a List.....	29
4.5.11 Resizing the List when the Window is Resized.....	29
4.5.12 Changing the Number of Fixed Columns.....	30
4.5.13 List Mouse Cursors.....	30
4.5.14 Tabwrap Navigation.....	30
4.5.15 Arrow Key Navigation.....	30
4.5.16 Refreshing a List.....	30
XIT_COLUMN4.6 COLUMNS.....	31
4.6.1 Disabled Columns.....	31
4.6.2 Enabled columns.....	32
4.6.3 Autoselected Cells in a Column.....	32
4.6.4 Read-Only Columns.....	32
4.6.5 Autoscroll Cells in a Column.....	32
4.6.6 Right-justified Columns.....	32
4.6.7 Password Columns.....	33
4.6.8 Platform and Well Columns.....	33
4.6.9 Centered Column Headings.....	33
4.6.10 Fonts for Column Headings.....	34
4.6.11 Icons in Column Headings.....	34
4.6.12 Multiline Column Headings.....	35
4.6.13 Platform and Well Headings.....	36
4.7 CELLS AND ROWS.....	36
4.7.1 Selected Rows and Enabled Rows.....	36
4.7.2 Colors Per Cell.....	37
4.7.3 Fonts Per Cell.....	37
4.7.4 Cell Range Selection.....	38
4.7.5 Putting Icons in Cells.....	40
4.7.6 Putting Bitmaps in Cells.....	41
4.8 FORMS.....	41
4.9 EDIT FIELDS.....	42
4.9.1 Disabled Edit Fields.....	42
4.9.2 Enabled Edit Fields.....	42
4.9.3 Autoselected Edit Fields.....	42
4.9.4 Read-Only Edit Fields.....	42
4.9.5 Autoscroll Edit Fields.....	43
4.9.6 Right-justified Edit Fields.....	43
4.9.7 Password Edit Fields.....	43
4.9.8 Platform and Well Edit Fields.....	43
4.9.9 Edit Field Buttons.....	43

4.9.10 Using <code>XI_ATR_FOCUSBORDER</code>	44
4.9.11 Multiline Edit Fields.....	44
4.9.12 Edit Field Fonts.....	46
4.10 GROUPS.....	46
4.11 CONTAINERS.....	46
4.12 BUTTONS.....	47
4.12.1 Types of XI Buttons.....	47
4.12.2 Using XVT Buttons.....	48
4.12.3 Disabled Buttons.....	48
4.12.4 Enabled Buttons.....	48
4.12.5 Icon and Bitmap Buttons.....	48
4.12.6 Radio Buttons.....	50
4.12.7 Check Boxes.....	50
4.12.8 Tab Buttons.....	50
4.12.9 Default Button.....	52
4.12.10 Drawing in Buttons.....	52
4.13 STATIC TEXT.....	52
4.13.1 Right-justified Static Text.....	53
4.13.2 Enabled/Disabled Static Text.....	53
4.13.3 Fonts for Static Text.....	53
4.14 RECTANGLES.....	53
4.15 LINES.....	53
4.16 WORKING WITH XVT/CH.....	54
4.17 SUMMARY.....	54
5	
DEFINING XI OBJECTS.....	55
5.1 OBJECT DEFINITION STRUCTURES.....	55
5.2 DEFINING AN INTERFACE OBJECT.....	56
5.3 DEFINING FORMS.....	57
5.4 DEFINING EDIT FIELDS.....	57
5.5 DEFINING LISTS.....	58
5.6 DEFINING COLUMNS.....	59
5.7 DEFINING CONTAINERS.....	60
5.8 DEFINING BUTTONS.....	60
5.9 DEFINING GROUPS.....	61
5.10 DEFINING STATIC TEXT.....	61
5.11 DEFINING RECTANGLES.....	61
5.12 DEFINING LINES.....	62
5.13 SUMMARY.....	62
6	
CREATING AN XI INTERFACE.....	63
6.1 SIZING THE INTERFACE.....	63
6.2 INSTANTIATING THE INTERFACE.....	64
6.3 HOOKING IT UP TO XVT.....	64
6.3.1 Programming the Task Window's Event Handler.....	65
6.3.2 Connecting XI Interfaces to XVT.....	65
6.3.3 Approach 1: Let XI Create the Window For You.....	65
6.3.4 Approach 2: Create the XVT Window with <code>xi_event</code> as Event Handler.....	66
6.3.5 Approach 3: Create the XVT Window with Your Own Event Handler.....	66
6.3.6 Putting an XI Interface in the Task Window.....	66
6.4 SUMMARY.....	67
7	
XI EVENTS.....	68

7.1 XI EVENT HANDLERS.....	68
7.2 RESPONDING TO XI EVENTS.....	69
7.3 REFUSING XI EVENTS.....	71
7.4 XI FOCUS MODEL.....	73
7.4.1 Basic Focus Rules.....	74
7.5 EVENT CATEGORIES.....	75
7.5.1 Interface Events.....	76
7.5.2 List Events.....	76
7.5.3 Form Events.....	79
7.5.4 Button Events.....	79
7.5.5 Focus Events.....	79
7.5.6 Special Events.....	80
7.6 THE XI_EVENT STRUCTURE.....	81
8	
USING XI OBJECTS.....	83
GETTING AN OBJECT POINTER	
8.1 GETTING AN OBJECT POINTER.....	83
<i>getting an object from an event</i> 8.1.1 <i>Getting an Object from an Event Structure</i>	83
8.1.2 <i>Using a Control ID</i>	84
<i>children of objects</i> 8.1.3 <i>Getting an Object's Children</i>	84
<i>parent of objects</i> 8.1.4 <i>Getting the Parent of an Object</i>	85
8.1.5 <i>Getting the Object With Focus</i>	85
8.1.6 <i>Making a Pseudo-Object</i>	85
8.2 USING EDIT FIELDS.....	86
8.2.1 <i>Being Notified of Typing in an Edit Field</i>	86
8.2.2 <i>Filtering characters</i>	86
8.2.3 <i>Validating Edit Field Text</i>	87
8.2.4 <i>Changing Edit Field Attributes</i>	88
8.2.5 <i>Changing a Single Attribute</i>	88
8.3 USING FORMS.....	89
8.3.1 <i>Validating the Contents of a Form</i>	89
8.3.2 <i>Interfacing to Databases When Using a Form</i>	89
8.3.3 <i>Setting the Keyboard Navigation Sequence</i>	92
8.4 USING LISTS.....	93
8.4.1 <i>Record Handles</i>	93
8.4.2 <i>How XI Manages the Record Handle Array</i>	94
8.4.3 <i>Managing Records</i>	94
8.4.4 <i>Displaying Text</i>	98
8.4.5 <i>Processing User Input</i>	98
8.4.6 <i>Responding to Focus Movements</i>	99
8.4.7 <i>Updating Databases When Using a List</i>	100
8.4.8 <i>Scrolling the List</i>	102
8.4.9 <i>Changing List Attributes</i>	105
8.4.10 <i>Changing the List Size</i>	106
8.5 USING CELLS.....	107
8.5.1 <i>Cell Request Events</i>	107
8.5.2 <i>Making Cell Objects</i>	108
8.5.3 <i>Being Notified of Typing in a Cell</i>	109
8.5.4 <i>Validating Cell Text</i>	109
8.6 USING ROWS.....	110
8.6.1 <i>Responding to Record Request Events</i>	111
8.6.2 <i>Making Row Objects</i>	111
8.6.3 <i>Deleting a Row</i>	112
8.6.4 <i>Inserting a Row</i>	112
8.6.5 <i>Validating the Contents of a Row</i>	113
8.7 USING COLUMNS.....	113

8.7.1 Getting a Column Object.....	114
8.7.2 Changing a Column's Heading.....	114
8.7.3 Changing the Width of a Column.....	114
8.7.4 Changing Column Attributes.....	114
8.7.5 Column Events.....	115
8.8 USING GROUPS.....	116
8.8.1 Validating a Group of Edit Fields or Cells.....	116
8.9 USING BUTTONS.....	117
8.9.1 Changing Button Attributes.....	117
8.9.2 Checking Radio Buttons and Check Boxes.....	118
8.10 USING STATIC TEXT.....	118
9	
MANAGING APPLICATION DATA.....	119
9.1 ASSOCIATING RECORD DATA WITH AN OBJECT.....	119
9.2 USING TREE MEMORY FOR APPLICATION DATA.....	121
10	
MEMORY ALLOCATION.....	122
10.1 PERFORMANCE CONSIDERATIONS.....	123
10.2 AUTOMATIC FREEING OF TREE MEMORY.....	124
10.3 DEBUGGING TREE MEMORY.....	124
11	
MODIFYING AN XI INTERFACE.....	126
11.1 ADDING OBJECTS.....	126
11.2 DEFINING AN OBJECT.....	126
11.3 INSTANTIATING AN OBJECT.....	127
11.4 ADDING A COLUMN.....	127
11.5 ADDING AN EDIT FIELD.....	128
11.6 DELETING OBJECTS.....	128
11.7 DELETING A COLUMN.....	129
11.8 DELETING AN EDIT FIELD.....	129
11.9 RESIZING AN XVT WINDOW.....	129
12	
INTEGRATING XI WITH XVT APPLICATIONS.....	131
12.1 USING XVT CONTROLS.....	131
12.2 DRAWING GRAPHICS.....	132
12.3 MENUS.....	132
APPENDIX A	
THE XI EXAMPLE.....	133
INDEX.....	137

Table of Figures

FIGURE 1 - LAYERS UPON LAYERS.....	2
FIGURE 2 - WHAT IS PROVIDED BY XVT AND WHAT IS PROVIDED BY XI.....	4
FIGURE 3 - EXAMPLE OF AN XI INTERFACE.....	7
FIGURE 4 - TYPES OF OBJECTS IN AN XI INTERFACE.....	8
FIGURE 5 - EVENTS GENERATED IN RESPONSE TO A MOUSE CLICK ON A CELL.....	10
FIGURE 6 - INTERFACE DIAGRAM.....	13
FIGURE 7 - INTERFACE DEFINITION.....	14
FIGURE 8 - ADDING A LIST DEFINITION.....	14
FIGURE 9 - EXAMPLE OBJECT DEFINITION TREE.....	15
FIGURE 10 - COMPARING FORM UNITS AND PIXELS.....	17
FIGURE 11 - A VIRTUAL INTERFACE.....	20
FIGURE 12 - XI INTERFACE WITH CLOSE BOX, SCROLL BARS AND SIZING.....	24
FIGURE 13 - LIST WITH A SCROLL BAR BUTTON.....	28
FIGURE 14 - A PLATFORM COLUMN.....	33
FIGURE 15 - MULTILINE COLUMN HEADINGS.....	36
FIGURE 16 - A SELECTED ROW.....	37
FIGURE 17 - A RANGE OF SELECTED CELLS.....	38
FIGURE 18 - AN EDIT FIELD BUTTON WITH DROP DOWN LIST.....	44
FIGURE 19 - A MULTILINE EDIT FIELD.....	45
FIGURE 20 - ICON BUTTONS.....	49

FIGURE 21 - TAB BUTTONS.....	51
FIGURE 22 - FOCUS FLOW AND OBJECT HIERARCHY.....	75
FIGURE 23 - EDITING A DATABASE RECORD IN A FORM.....	89
FIGURE 24 - RECORD HANDLES FOR A DATABASE LIST.....	93
FIGURE 25 - HANDLES TO AN IN-MEMORY ARRAY.....	95
FIGURE 26 - HANDLES TO A LINK LIST STRUCTURE.....	96
FIGURE 27 - HANDLES TO DATABASE RECORDS.....	96
FIGURE 28 - FREEING TREE MEMORY.....	123
FIGURE 29 - THE MEMORY LIST.....	133
FIGURE 30 - THE LINKED LIST.....	134
FIGURE 31 - THE LINKED LIST CHANGE DIALOG.....	134
FIGURE 32 - THE EMPLOYEE LIST.....	135
FIGURE 33 - THE ADD EMPLOYEE DIALOG.....	135
FIGURE 34 - THE CHANGE EMPLOYEE DIALOG.....	136
FIGURE 35 - THE DELETE EMPLOYEE DIALOG.....	136
FIGURE 36 - THE SYNCHRONIZED LISTS.....	137

1

Introduction

Each part of the *XI Programming Manual* provides a different perspective of the XI tool kit. The purpose of the *XI Programmer's Guide* is twofold. It is meant 1) to give you a comprehensive look at the tool kit's functionality, and 2) to lay down a practical foundation in XI programming techniques.

If your time is limited and you want to read the minimum number of manual pages before writing a serious application with XI, we recommend that you start with the *XI Programmer's Guide*. The *XI Programmer's Guide* is arranged according to the steps you will need to take if you were writing an XI application. As you are reading the *XI Programmer's Guide*, you should look at the sample application shipped with the tool kit. Much of this code may be usable as a starting point for your own application. The *XI Programmer's Reference* will give you more precise information than provided in the *XI Programmer's Guide*. You should look there to find exact specifications of XI structures, events and functions. Pay particular attention to the information about record handles in the chapter *Using XI Objects*.

If your time is severely limited, you may want to learn XI just by looking at the sample program. However, you are likely to miss many of the details and options available with that method. In any case, you should still look over the information about record handles in the chapter *Using XI Objects* since this is the most commonly misunderstood aspect of XI.

This manual assumes that you are already familiar with basic GUI programming concepts such as events, event handlers, windows and controls. In addition, if you are new to programming with XVT, we recommend that you read parts of the XVT documentation before moving to XI. In particular, read the XVT documentation that addresses compiling and linking applications on each XVT platform, and the documentation that shows you how to program the **main** function and the task window's event handler. You may also want to read the XVT documentation about fonts and font metrics, drawing text and basic XVT event processing, as well as a quick review of the graphics primitives, just to get an idea of what is available from XVT.

1.1 What is XI?

XI is a library. It is a set of 'C' functions and data structures that allows you to create user interfaces. XI is not a new language, resource compiler or editing tool. It is merely a function library and set of data structure definitions. Although written in C, XI works easily in C++ programs.

You use XI to create portable applications that display and edit data in a form or spreadsheet-style list. XI offers a variety of ready-made objects with predefined behaviors that can give an application a higher level of functionality and more sophisticated look and feel than you can have with XVT alone. Because XI was built using XVT, both the XI source code and your code written using the XI library will be portable across all platforms supported by XVT. Of course, you can also take full advantage of any XVT feature.

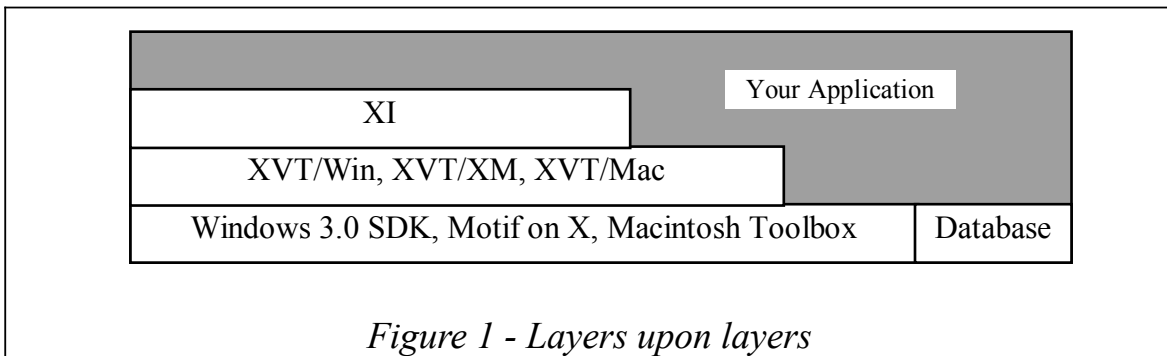
Therefore, you use XI to expand the number tools you can use to create a portable application using XVT.

In this chapter you will find a description of the application programming environment you will encounter when building an application with XI, XVT and native tool kits. In addition, we will explore the steps involved in constructing an XI application using XI and XVT. In the next chapter, you will find an overview of XI objects, including spreadsheets and forms. There, you will also encounter a discussion of the events XI objects can generate, and an explanation of the event handler that you will need to write in order to respond to those events. Together, these chapters will give you a general understanding of the XI library, and you will be ready to program an XI application using the instructions found in later chapters.

1.2 Layers Upon Layers

When programming an XI application, there will be three or more tool kits underneath your code on each machine you are supporting. For example, if you are writing an XI application for Microsoft Windows, you will encounter the XI tool kit for Windows, XVT for Windows and the Windows SDK. These tool kits are layered, with the “native” tool kit on the bottom of the stack, XVT for that platform on top of the native tool kit and XI on top of XVT. When we say that a tool kit “sits on top” of another tool kit, such as XI on top of XVT, we mean that the tool kit on top uses the functions of the tool kit below it, but not vice-versa. XI can call XVT functions, but XVT doesn’t use any of the symbols defined by XI. In addition, we often say that a programmer can “drop down a level” to access the functionality of a tool kit beneath it. Since your code is on top of all of the tool kits in a layered API, you can access any layer anywhere in your program.

Look at the following picture for a summary of the tool kits you might encounter. Following the picture is a description of the functionality provided by each of the layers.



1.2.1 XI

As you will see in later chapters, an XI application is composed of XI interfaces which contain XI objects. The objects are the spreadsheets, forms, edit fields, buttons, lines, and rectangles found in an XI interface. An XI interface is XI's object that contains information for a specific window. There is a one to one correspondence between XVT Windows and XI interfaces. To define an interface, you will be using certain XI functions. Other XI functions contain all of the functionality you need to write a XI application. You will encounter these functions when you are instantiating and manipulating the objects you’ve created. For more information on XI objects and interfaces, see the next chapter, *An XI Interface*. For more information on functions to manipulate objects see *Using XI Objects*. For more information on defining and creating an interface, see *Defining and Creating an Interface*.

1.2.2 XVT

The purpose of XVT is to provide portability. XVT's job is to unify the separate platforms under one programming interface. Because XI was written using the XVT functions and data structures, it achieves its portability as a direct result of using XVT. In addition, you may want to use the functionality provided by XVT. For example, you may want to have menus and modal dialog boxes or pie charts and graphs drawn with XVT drawing primitives. These are provided at the XVT level. It is important to remember that XVT forms a substrate upon which XI is built and your applications will be built using both layers. When creating more sophisticated applications, you will need to know how to program using parts of XVT.

1.2.3 The Native System

In general, programming within the native tool kit means that your code will be non-portable. However, there are times when programming with only XI and XVT is insufficient because you need more specific control over a platform than the portability layers are able to provide. If this is the case, then you'll need to read the documentation in the section titled *Installing and Using XVT*, which comes with the XVT tool kit. There is a separate section for each XVT platform.

The *Installing and Using XVT* sections will explain how to get a hold of the events and call the native tool kit functions that are necessary to perform low level programming in the native tool kit. Once again, it is recommended that you avoid this if possible, because you will lose the portability benefits of XVT.

If you must program in the native tool kit, we recommend that you get the complete set of references for that particular platform.

1.3 Constructing an XI Application

In the remaining half of this chapter, we will outline the steps you'll need to take when constructing an XI application. By "XI application" we mean an application which uses XI to create a spreadsheet, form, or other objects in a XVT window. It is possible that this form or spreadsheet is only a small portion of your application, but the term is helpful when discussing the environment you'll be programming in for the portion that uses XI. Keep in mind that XI never precludes your application from using XVT calls. For example, your application may draw other graphics in the same window containing an XI object. When manipulating XI objects, your application would use XI functions. However, when drawing graphics your application would call XVT directly.

When constructing an application using XI, you will need to supply some XVT code in addition to the code you write with XI. The following picture summarizes what kinds of things you will need to provide at the XVT layer and what kinds of things are provided at the XI layer.

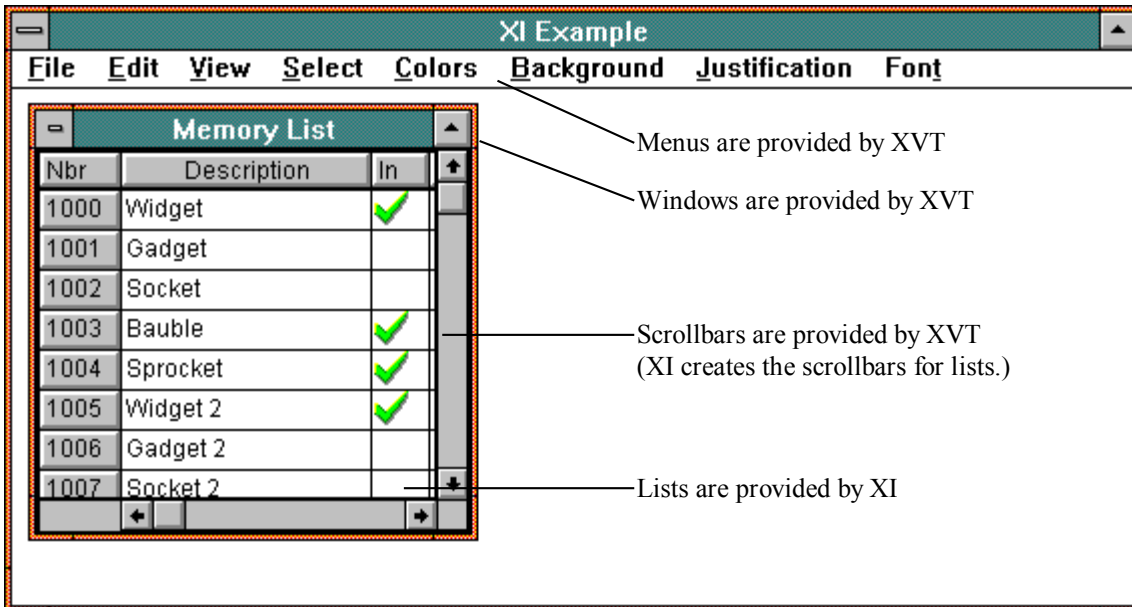


Figure 2 - What is provided by XVT and what is provided by XI

As you can see in the picture above, an XI application is actually a hybrid application. Some parts of the application are written using XVT and some parts are written using XI.

1.3.1 XI Programming

As you'll see in the next chapter, an XVT window can have an XI interface which contains XI objects. (When using XI, the XI interface is XI's object for the XVT window. An XI interface can be thought of as the XVT window.) The objects in a window are defined using functions and are instantiated at runtime (as opposed to being part of a resource file). In addition to defining and creating an interface, you will need to write event handlers to process the events generated by the operation of the interface. Using XI functions, you will manipulate XI objects in response to those events. You may also use XI's application data for objects and XI's tree memory management.

1.3.2 XVT Programming

To create any application with XI, you will need to do some XVT programming. As with any XVT programming effort, it is important to go back and forth between the hardware platforms you'll be supporting. Follow the rule, "port early and often."

For the most basic application using XI, you will need to do several things in XVT. These are explained in detail in *Creating an Interface*, but for now, here is a summary of what you'll need to do:

1. Include **xi.h** instead of **xvt.h** in your C module.
2. Write a **main** function.
3. Write a task window event handler. In this function you will need to place a call to **xi_init** which tells XI to initialize itself.

All of the XI examples use at least the three elements of XVT programming detailed above. Some examples use more elaborate approaches. Several approaches are explained in *Creating an Interface*.

1.4 Summary

After reading this brief introduction to the XI programming environment, you are ready to start writing an application with XI. You learned that XI is layered on the top of other tool kits, and that you can program to any layer. Of course, you will need to install XI, XVT, the native tool kit and an appropriate 'C' compiler for each platform you are supporting — not an easy task on some systems.

In the next chapter, *An XI Interface*, we will look more closely at an XI interface and its event handler.

2

An XI Interface

This chapter defines what an XI interface is and describes the objects an XI interface can contain. In the description of XI objects, you will see how XI objects are similar and different from objects in the object-oriented sense. After encountering the discussion of XI interfaces and objects, you will learn that XI is event-driven since it “sits on top” of XVT and responds to XVT events. In addition, XI generates its own events to which you will need to respond in an event handler function.

An XI interface is a window containing any number of user interface objects. Most XI objects are operable by the user and will invoke some action in the application. (Some XI objects are display-only.) Together the objects inside the interface behave in a coherent fashion to achieve the objectives of the program. You can think of these objects as analogous to controls in the XVT parlance or widgets in the vocabulary of some native tool kits. For an overview of the kinds of objects an XI interface can hold, all types of XI objects are found in the following picture.

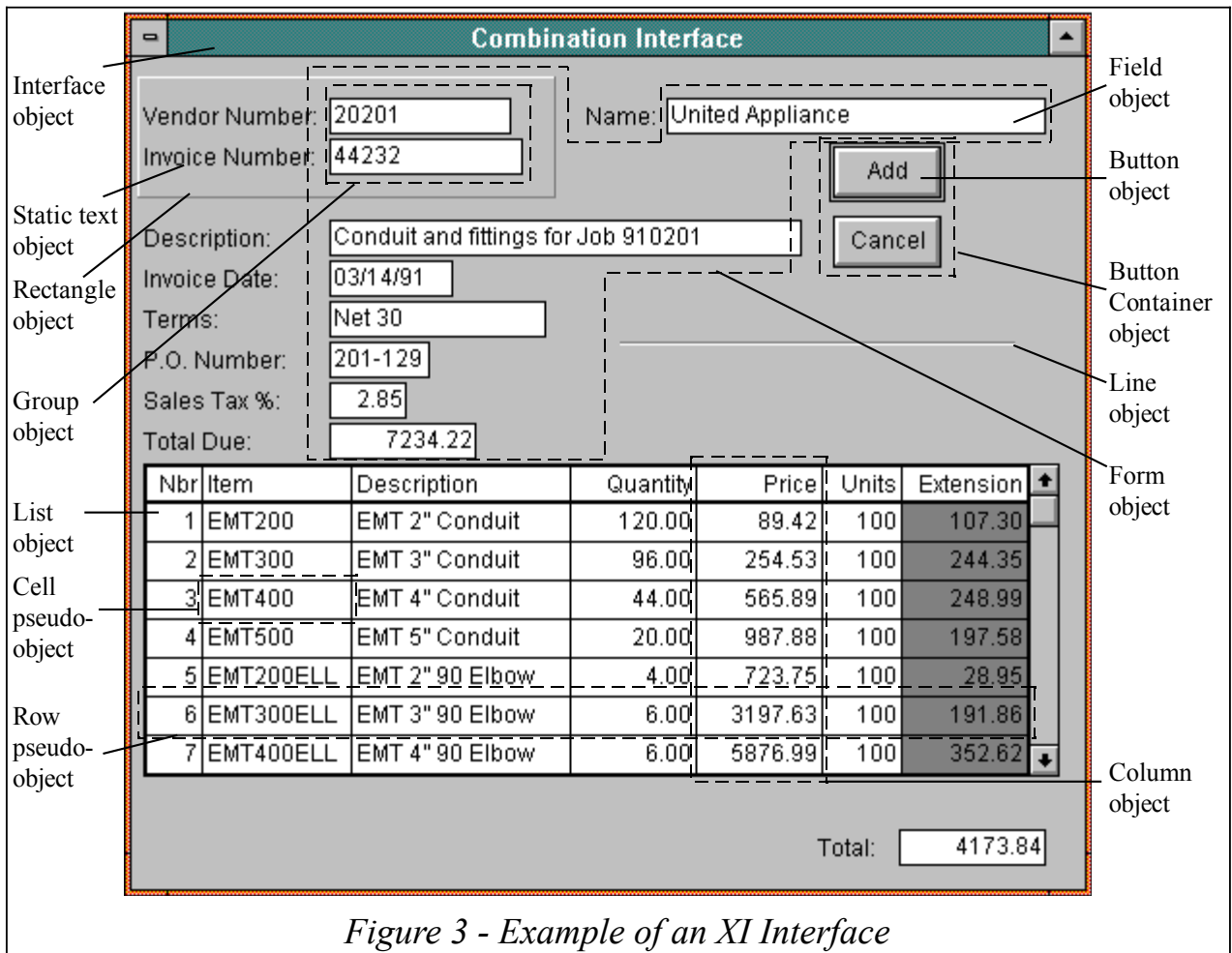
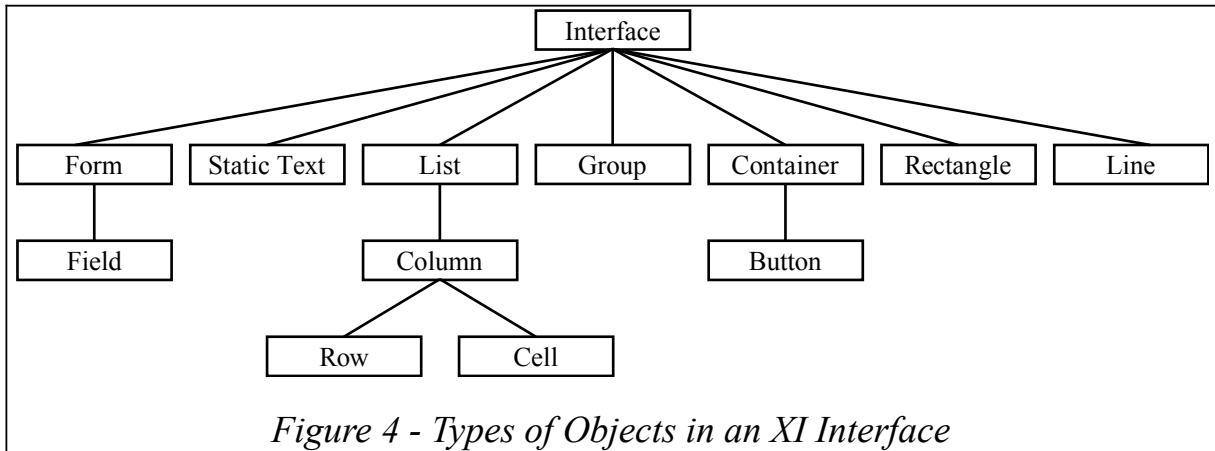


Figure 3 - Example of an XI Interface

XI Interface As you can see, an XI interface can hold any number of XI objects such as forms, lists, containers and groups. These objects might in turn hold other XI interface objects such as edit fields, columns and buttons. (Edit fields are called edit controls in MS-Windows.) In this sense, the XI object structure is hierarchical with the interface at the top of the tree, composite objects such as forms, lists, containers and groups on the next level, and edit fields, columns and buttons on the lowest level. In the following diagram, you will see an example tree showing how XI interface objects are related to one another. Imagine that this tree corresponds to an interface that has a form with one edit field, one line of descriptive text, a list with one column and one row, a group (of either edit fields or columns), a rectangle, a line, and a container with one button. In the real world this interface would not be a very practical, but it is useful for illustrating the parent-child relationships between XI objects. When we describe XI objects in more detail in the next section, keep this diagram in mind. Note that buttons can be direct children of an interface if you do not wish to put them in a container.



2.1 Summary of XI Objects

When you look at the diagram "*Types of Objects in an XI Interface*", you can see the relationships XI objects have to one another. Every XI object must be created within an interface object, and therefore each XI object must have an interface as one of its ancestors. This makes sense from the user's point of view because whenever you create a control, you must have a window to put it in. You can also see that lists, forms, groups, containers, buttons, rectangles, lines, and static text are children of the interface while edit fields, buttons and columns are grandchildren. The XI objects in the object hierarchy that are children of the interface object are summarized following this paragraph. Since list, form and container objects have children of their own, we call them *composite objects*. Columns are children of lists, edit fields are children of forms, and buttons are children of containers:

Lists: XI's spreadsheet-style lists can be scrolled by the user and the user can edit each cell in the spreadsheet if your application permits it. In an XI list, columns are the only child objects of the list. Cells and rows are "pseudo-objects" because they are not actually instantiated when the interface is created. Nevertheless, you can treat them like real objects and "use" them to get and set text, get and set attributes, set focus, and so on.

Forms: Forms contain any number of edit fields that can be edited. Edit fields are child objects of forms. (Edit fields are like edit controls in MS-Windows.)

Containers: Containers serve to arrange buttons. You can either stack buttons on top of one another, or arrange them end to end. Buttons can also be arranged in a grid. Buttons are child objects of containers. In addition, if a container contains radio buttons, the container serves to group the radio buttons together so that only one will be checked at a time. Buttons can also be children of the interface.

Groups: You can have groups of edit fields or columns. In XI, groups are there to make it easier to validate data entry for associated edit fields or columns. Groups have no appearance in the window. To group controls visually, use XI rectangle controls.

Static Text: Static text is used to label objects on the interface.

Rectangles: Rectangles are used to visually group objects on the interface.

Lines: Lines are used to visually separate objects on the interface.

2.1.1 Objects

From the previous description of interfaces and the objects they can contain, you saw what objects look like and have some ideas about what they do. We'll talk more about the look and feel of XI objects in later chapters, but in this section, we want to compare XI objects to those found in object-oriented programming

environments. This section is written mainly for those who are familiar with object-oriented programming to help you understand how XI objects are similar to and different from the objects you are used to. If you are not familiar with this style of programming, the following discussion may still be of some value to you as there are some properties of XI that are object-oriented.

We will begin this section with a brief description of the essential properties of objects in the object-oriented sense. These properties are encapsulation, inheritance, instances and identity. Following this, we will explain how XI objects share some of these properties and not others.

2.1.1.1 Object-Oriented Programming Concepts

Encapsulation: Very briefly, object-oriented programmers say that an object is a software “package” that contains its own private data and the appropriate code (or methods) for managing the behavior of the object. Through a programming technique called “encapsulation”, the object’s data is accessible only through the methods bound with it.

Instances: An object is made by creating an “instance” of a class (its type), or “instantiating” it. Memory is allocated for the corresponding data structures found in the class definition, and all of the values in the data structures are initialized to their default values. In addition, a pointer or handle to the class definition and a unique object identifier may be stored in the instantiated object.

Inheritance: The class or type of an object may be derived from a previously defined class using “inheritance”. This allows for a hierarchy of classes which build from simpler to more complex definitions of data and methods. Each class inherits the data and methods of the class it is derived from, but may add or enhance that class definition.

Identity: Every instantiated object has a unique handle or pointer associated with it that distinguishes one instantiated object from another.

2.1.1.2 A Comparison of XI to Object-Oriented Programming

XI objects share some of the above concepts but not others. XI objects do have their own private data and methods that operate on that data, but the methods are not bound with the data. As you’ve seen before, XI objects are members of an object hierarchy, but that hierarchy is not a class hierarchy, it is a hierarchy of instantiated objects. XI children objects cannot inherit their parent’s attributes.

As in true object-oriented programming, XI objects are instantiated from a description of the object. However, this description is not a class. Instead, an XI object is instantiated from an object definition structure. The objects have a **type** field that determines which methods (functions) are valid and how they will work for that object.

Like the objects in object-oriented programming, XI objects also have unique identity. This identity is of the form of a pointer to an object, and this pointer can be used by all of the XI functions that can manipulate the object. For example, XI has a function called **xi_set_text**, which will set the text of an object. Depending on the type of object it is passed, **xi_set_text** will know to set the title of the window if the object is an interface or set the heading text of a column if it is a column. In this way, XI has a “regular” and “orthogonal” programming interface since you can pass an pointer to a function and the function knows what methods to use based on the type of object it encounters.

2.1.2 Events

eventsLike every GUI programming environment, XI uses events to communicate with the application. A GUI tool kit will send an event to the application for one of two reasons: either to inform the application

that the user has manipulated a user-interface object, or to ask the application for more information. To illustrate, let's compare two kinds of events an XI application might receive concerning a list object. Let's suppose that an application just created a list. In order to draw the list, XI must send the application **XIE_CELL_REQUEST** events requesting it to supply the text to be displayed in the cells of the list. This is an example of an event asking for information. In contrast, let's suppose that the cursor is in the cell of the list and the user presses the down arrow key. To notify the application of this action, XI will send it an **XIE_OFF_CELL** event to give the application a chance to check to see if the contents of the cell represent valid data. This is an example of an event informing the application of the user's action. Keep in mind that upon receiving an event, the application doesn't have to respond. It can ignore or refuse it.

2.1.2.1 Event Flow

As you saw in the introduction, an XI application is built on top of several layers of other tool kits. Recall that underneath everything is the underlying window system and tool kit such as Microsoft Windows, or the X window system with Motif. We call this layer the "native tool kit". Built on top of the native tool kit is XVT, on top of XVT is XI, and on top of everything is your application.

When an event is generated it usually originates down at the lowest level (we say usually because it is not always the case). Thus, if the user clicks the mouse, the native window system will send an event to XVT notifying it that the mouse was clicked. XVT will then translate that into a portable mouse click, and pass the event on to the next layer which is XI. XI will receive the portable mouse click, process it, and perhaps route it to an object that will interpret it to mean that the user has manipulated the object in some way. After deciphering the event, XI may generate one or more higher-level events that it passes on to the application. In the following illustration, you will see the flow of events for a mouse click on a cell in an XI list when the user is moving to a new location in the spreadsheet.

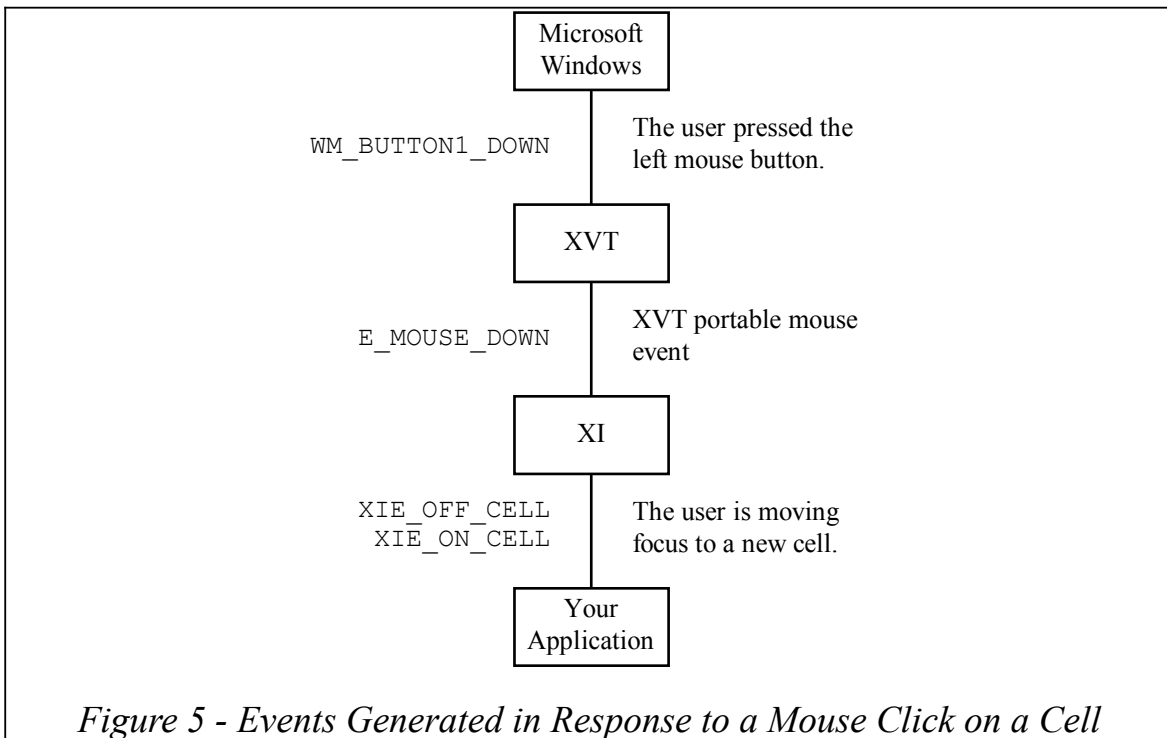


Figure 5 - Events Generated in Response to a Mouse Click on a Cell

As you can see in the event flow diagram, the difference between the events received by XI and those of most GUI tool kits is that XI events are higher-level. By higher-level we mean that the events can provide more information about what the user is doing, or what the application needs to do next. For example, XI might send an event asking the application to supply text for several data records or to inform it that the cursor has moved to another cell in a column. In contrast, Microsoft Windows might send an event (in Windows, they are called messages) requesting the maximum dimensions of a window or to indicate that

the mouse has moved. The concepts are the same. In both cases, an application is being notified of an action by the user or is being asked to supply information. The difference lies in the amount of information the application can receive or needs to give.

2.1.3 XI Event Handler

XI notifies the application of events by calling a special function supplied by the application called an event handler function. Each XI interface has an event handler function that handles the events generated whenever any object inside the interface is being manipulated by the user, or needs information from the application. Whenever an event occurs, the event handler function corresponding to the affected interface is called. An event structure is passed to the event handler. This structure tells the event handler that the event has happened and informs it of what action needs to be taken by the application to respond to the event.

An application can have more than one event handler. In particular, it can have one event handler for each instance of an interface. For example, you could instantiate an interface twice using the same objects, except associate a different event handler with each instance. On the other hand, you may have one event handler for all of the interfaces and use application data to differentiate them. As you might suspect, an XI application will spend most of its time responding to events in event handlers, and therefore, the majority of your time spent writing the XI portion of your application will be spent coding event handlers. For help with writing event handlers, look in *XI Events* and *Using XI Objects* found later in this guide.

It is important to note that there is one and only one event handler per interface. If you want event handlers for each object, you will need to use the application data for the objects to hold that event handler pointer and then write the code to “dispatch” the events from the event handler for the interface. event handler per object

2.2 Summary

In this chapter, you saw what an XI interface looks like and have some idea about the kinds of objects it can contain. You also saw the mechanism by which these objects generate events. In the next chapter, we will examine in detail the mechanism of creating an XI interface.

3

Creating an Object Definition Tree

In XI, before you can instantiate an interface, you must define an object definition tree. This tree of structures tells XI what objects to create and how they relate to one another. The focus of this chapter is constructing an interface hierarchy to conceptualize how the objects in an interface are related and then using certain functions to define the interface. When you have finished defining an interface, you will have an object definition tree. After taking an overview of the object definition tree, we will look at the options available when creating each specific type of object in the tree.

It is useful at this point to understand the process and mechanism of creating XI objects. When creating an XI interface, first you will create a tree of structures that define the objects for the entire XI interface. Remember that the XI interface is analogous to an XVT window that contains XVT controls. After creating these structures, you will call `xi_create` to actually create the XI interface and its contained XI objects. Instantiating the tree of objects by calling `xi_create` is the topic of the chapter, *Creating an XI Interface*.

Note: The object definitions can be used to add more objects to an interface after it has been initially created. However, it is most common to define all of the objects before creating the interface.

Creating the tree of structures that define an XI interface is moderately complicated, so we created the XI convenience functions. These convenience functions take arguments that:

- define the look and feel of an object
- set its control ID
- initialize pointers to parent or children definitions
- set dimensions

- define its place in the tabbing sequence
- allocate the space for the object definition and insert it into the hierarchy

Every XI object has a control ID. A control ID is a unique integer by you. Control IDs have the same purpose as the control IDs used in the XVT tool kit. We will explain more about control IDs later.

Convenience functions do not set all fields in the structures that define an object. Only the most basic information is passed as parameters to the convenience functions. However, convenience functions always use memory that has been allocated and cleared. (When the memory is cleared, all bytes are set to zero.) After calling the convenience functions, it is possible to set fields that were not set by the convenience functions. In this fashion, we have created a completely extensible programming interface. If we wish to add a new feature to XI, we can add a field to the object definition structure. When we do this, if the field is cleared (sometimes we say that the field contains zero bytes), then we give the object the same behavior that it always had. Therefore, we can add features to XI without impacting your existing code.

Another way to think about it is that the most common information is conveyed to XI via parameters to the convenience function. To set advanced parameters, you set fields in the object definition structure returned by the convenience function.

The following code shows an example of calling a convenience function, then setting a field in the object definition structure after the call to the convenience function:

```
XI_OBJ_DEF* btndef;

btndef = xi_add_button_def( cntrdef, DELETE_ALL_CID, NULL,
                           XI_ATR_ENABLED | XI_ATR_VISIBLE,
                           "Delete All Recs", ADD_ONE_CID );
btndef->v.btn->fore_color = COLOR_RED;
```

3.1 Designing an Interface Hierarchy

When using XI for the first time, it is helpful to draw a picture representing how the objects that you will be defining and instantiating are related to one another. In XI, it is essential to know which objects are parents and children of other objects so that when you define the object definition tree, you will know which convenience functions to call and what to pass them. In the following picture you will see a representation of the relationships of XI objects to one another.

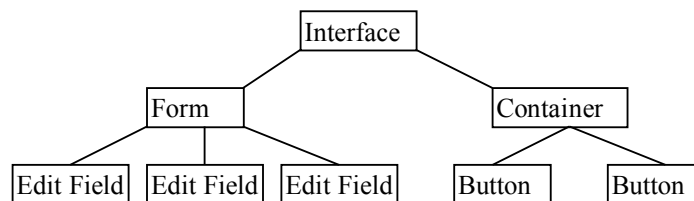


Figure 6 - Interface Diagram

3.2 Using the Convenience Functions

As we mentioned, the XI convenience functions are there to help you define the structures needed to describe XI objects and to build XI interfaces. In particular, these convenience functions will manage allocating memory for the definition tree and will fill in the fields of the structures needed to create an object definition hierarchy. The result of using the convenience functions is that you will have a pointer to an object definition hierarchy which you will pass to `xi_create` to instantiate the objects defined. The steps you'll need to take to build an object definition tree using the convenience functions are outlined below.

To define an object definition tree, you start at the top, defining objects as you descend down the tree. Therefore, the first step is always defining the interface object by calling `xi_create_itf_def` with the appropriate parameters. `xi_create_itf_def` will create two structures that together define an interface. The structures it creates are an XI object definition (`XI_OBJ_DEF`) and an interface definition (`XI_ITF_DEF`) as show below. `xi_create_itf_def` returns a pointer to the `XI_OBJ_DEF` for the interface.



Figure 7 - Interface Definition

When you use convenience functions other than `xi_create_itf_def`, you will be attaching objects to an existing interface tree, or “adding” them. These functions are called to define the children or grandchildren of an interface definition, and require you to pass in a reference to the parent of the definition that you are adding. (Note: If you do not pass a parent, then you will need to pass the appropriate parent object to `xi_create`. This will add the object to an existing interface.) The convenience functions will add the object definition to the array of children objects of the parent. For example, if you are adding a list definition to an interface definition you would call `xi_add_list_def`, passing it the `XI_OBJ_DEF` you got when `xi_create_itf_def` returned. `xi_add_list_def` will create an `XI_OBJ_DEF` and `XI_LIST_DEF` as a child of the interface definition as shown below.

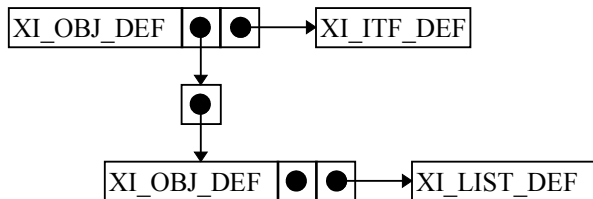


Figure 8 - Adding a List Definition

As you can see in the picture above, an `XI_OBJ_DEF` is created for each object definition. This is because it is the generic structure we use to store information common to all objects and keep track of relationships between them. Unique information is stored in a structure specific to an object. This is why you pass an `XI_OBJ_DEF` to the convenience functions and not an `XI_ITF_DEF` or an `XI_LIST_DEF`.

In addition to lists, you can place groups, containers, buttons, forms, rectangles, lines, and static text directly below an interface. Of the objects that can have an interface as a parent, containers, forms and lists will contain other objects. As you saw before, to add a list to an interface, you use the function `xi_add_list_def`. To add the other definitions, you use the functions `xi_add_group_def`, `xi_add_container_def`, `xi_add_form_def`, `xi_add_rect_def`, `xi_add_line_def`, and `xi_add_text_def`. Each of these functions returns an `XI_OBJ_DEF`. For containers, forms and lists, this `XI_OBJ_DEF` will be passed to the convenience functions used to define their children.

Here is some sample code that creates an object definition tree containing a list with two columns and a container with two buttons. A diagram of the resulting object definitions follows this code.

```

XI_OBJ_DEF*      itf_def;
XI_ITF_DEF*      itf_def_detail;
XI_OBJ_DEF*      list_def;
XI_LIST_DEF*     list_def_detail;
XI_OBJ_DEF*      column_def;
XI_COLUMN_DEF*  column_def_detail;
XI_OBJ_DEF*      container_def;
XI_OBJ_DEF*      itf;
XI_RCT           rct;

itf_def = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh, NULL,
                             "Show Columns", 0L );

```

```

itf_def_detail = itf_def->v.itf;
itf_def_detail->automatic_back_color = TRUE;
itf_def_detail->modal = TRUE;
itf_def_detail->use_whitespace = TRUE;
itf_def_detail->whitespace_right = 0;
itf_def_detail->whitespace_bottom =
    (int)xi_get_pref( XI_PREF_ITF_WS_BOTTOM );

list_def = xi_add_list_def( itf_def, LIST_CID, 0, 0, 8 *
XI_FU_MULTIPLE,
    XI_ATR_ENABLED | XI_ATR_VISIBLE
    | XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
    COLOR_BLACK, COLOR_WHITE,
    COLOR_BLACK, ADD_BTN_CID );
list_def_detail = list_def->v.list;
list_def_detail->scroll_bar = TRUE;

column_def = xi_add_column_def( list_def, SELECT_COL_ID,
    XI_ATR_SELECTABLE, 1,
    6 * XI_FU_MULTIPLE, 1, "Show" );

#if THREE_DIMENSIONAL != 0
column_def_detail = column_def->v.column;
column_def_detail->heading_platform = TRUE;
column_def_detail->column_platform = TRUE;
#endif

column_def = xi_add_column_def( list_def, NAME_COL_ID,
    XI_ATR_SELECTABLE, 2, 20 *
    XI_FU_MULTIPLE, 20, "Description" );
column_def_detail = column_def->v.column;
#if THREE_DIMENSIONAL != 0
column_def_detail->heading_platform = TRUE;
#endif
column_def_detail->center_heading = TRUE;

rct.top = 9 * XI_FU_MULTIPLE;
rct.left = 4 * XI_FU_MULTIPLE;
rct.bottom = 11 * XI_FU_MULTIPLE;
rct.right = 22 * XI_FU_MULTIPLE;
container_def = xi_add_container_def( itf_def, CONTAINER_CID, &rct,
    XI_STACK_HORIZONTAL, LIST_CID );

xi_add_button_def( container_def, ADD_BTN_CID, NULL,
    XI_ATR_ENABLED | XI_ATR_VISIBLE, "Show",
    CANCEL_BTN_CID );

xi_add_button_def( container_def, CANCEL_BTN_CID, NULL,
    XI_ATR_ENABLED | XI_ATR_VISIBLE, "Cancel",
    LIST_CID );

```

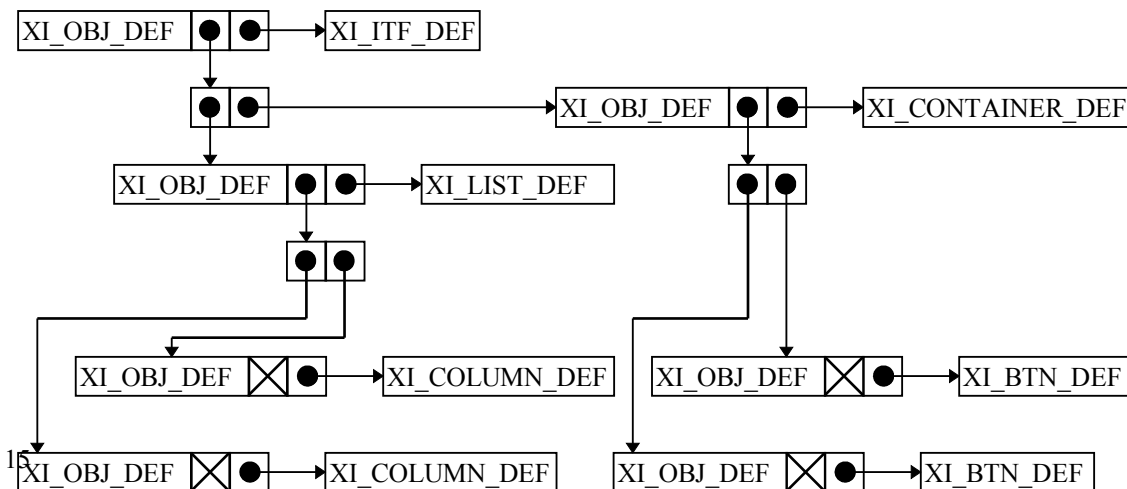


Figure 9 - Example Object Definition Tree

4

Characteristics of XI Objects

In Chapter 2, we gave a conceptual overview of an XI interface and the objects contained in it. In addition, you saw how XI events are generated and learned about the event handlers you'll need to write in order to respond to the events. In Chapter 3, you saw how XI expects you to create an XI interface definition tree.

In contrast, the focus of this chapter is the variety of options available for each type of XI object. Here, we will describe many of the possible appearances and behaviors each XI object can have. (For a complete list of the options for an object, see the *XI Programmer's Reference*.) After reading this chapter you will know many of the options you have for specifying object characteristics.

In addition to the descriptions of the look and feel of each type of object, this chapter provides code examples demonstrating the use of the features of XI. In some cases, portions of the programming examples contain information that is not properly introduced until later chapters. However, after you have familiarized yourself with XI programming, it is much more convenient to have the example of how to code a particular feature next to the description of the feature. The first time you read this chapter, you might wish to read it concentrating on the features available, and skip over the coding details. After you have read the following chapters, then you can then refer back to this chapter to examine the code more carefully.

As mentioned in previous chapters, the first thing you must do when programming with XI is create an object definition tree for the interface you want to instantiate. Each of these objects has a variety of parameters that are passed to the convenience functions. Also, there are a large number of other options that can be set for an object definition after it has been created by the convenience function.

When you are creating an interface, you will need to ask yourself these questions:

1. What objects do I want?

2. Where will each object appear in the interface (window)?
3. What other characteristics will each object have?

After reading this chapter, you will have a good idea about the possible answers to these questions.

To help you answer the second question, we will describe the coordinate system that XI uses.

4.1 Coordinate System

In XI, whenever you specify how big you want an object to be and where you want it placed on the screen, you do so in terms of “form units”. Form units are abstract units of measure that are related to the natural height and width of XI objects drawn on the native platform. In particular, the height of eight “form units” is equal to the height of an edit field, based on the font for the interface, plus an appropriate amount of white space above and below the edit field. The width of eight “form units” is the width of an average character, based on the font for the interface.

The following picture shows a comparison of pixels and form units for a sample edit field. This is a “zoomed in” view of an XI interface with a pixel grid overlaid on it.

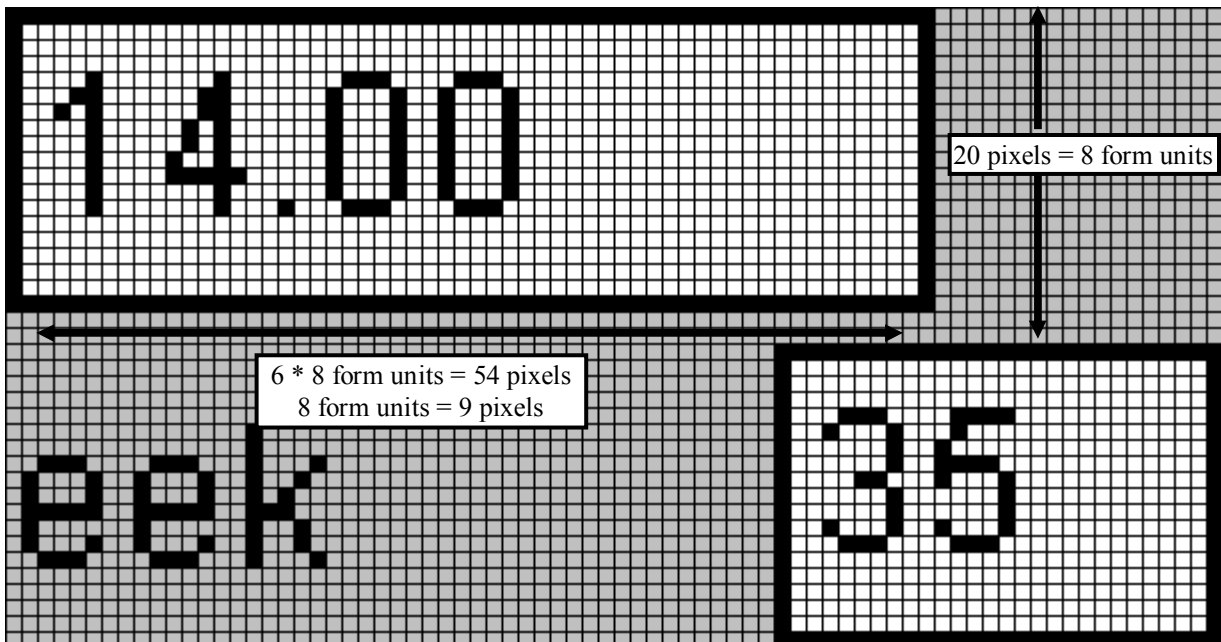


Figure 10 - Comparing Form Units and Pixels

As you can see, there is not an even number of pixels per form units. However, there is always an even number of pixels for each 8 form units.

The size of form units may vary greatly from system to system. For example, under windows, a normal font may be 15 pixels for each 8 form units. On a screen that is 480 pixels tall, this allows a maximum of 256 form units. On a character system, with 25 lines, each line is 8 form units, so the maximum is 200 form units.

The variation in font sizes can be a problem if you are using a pixel based coordinate system as opposed to a more abstract one like XI uses. Thus, the main reason XI uses form units as the basis of its coordinate system is greater portability of XI interfaces across systems, as you'll see in the following discussion.

4.1.1 Form Units -vs- Pixels

Most developers creating GUI applications are accustomed to using a pixel-based coordinate system to specify the size, shape and location of the objects they want to draw on the screen. For example, in XVT, you use a universal pixel-based coordinate system to describe where you want an object drawn. From these universal coordinates, XVT figures out which pixels to turn on and off on each native system.

While the precision of a pixel coordinate system is necessary for applications like drawing programs that need to display ovals and Bézier curves, when programming applications that need to layout edit fields, push buttons, radio buttons, check boxes, and multi-column lists, it is more convenient to use an abstract coordinate system that is described in terms of the natural size of the objects you'll be drawing. In addition, when you use a pixel-based coordinate system such as XVT, you can run into several problems porting your application to another platform. This is mainly due to platform differences in the number of pixels they use to draw native objects (such as buttons and edit fields). For example, on the Mac, an edit field might be 14 pixels high; on Motif, it might be 18; and on Windows, it might be 20. If you used pixels to describe the placement of objects, it would be difficult to predict what your application will look like on another system where the natural sizes of objects are different.

In XI, when you use form units for placing objects such as edit fields, you can imagine a grid where the height of the rows is the height of an edit field and the width of the columns is the width of a character. Then, when you specify that an edit field be placed on the second row (16 form units) down and the third column (24 form units) over, the edit field will be placed in an appropriate location on any of the XVT platforms.

For example, in XVT/CH, the edit field will be placed on the second line and third character from the left of the window. In Microsoft windows, the edit field will be placed in such a way that when you create another edit field on the third row and the third column over, the edit fields will be spaced an appropriate vertical distance from each other, and the left border of the two edit fields will line up.

In addition, using form units to specify the size and location of objects on the screen is more convenient than using a pixel-based system. When arranging edit fields on the screen, you can simply space them out by eight form units vertically, and the user interface would have the correct appearance on any of the systems supported by XI.

Perhaps you may wonder why you would want to use anything other than a multiple of eight form units. This allows for finer positioning without losing the advantages of a form unit coordinate system. However, if you plan on porting your application to XVT/CH, always use form units in multiples of eight.

4.2 Control IDs

As we mentioned, every XI object has a unique control ID. Control IDs are unique integers defined in header or C files that you write. Here is an example of definitions of control IDs from "lstdb.c":

```
/* Control IDs for the employee list */
#define ITF_CID 1
#define LIST_CID 2
#define CONTAINER_CID 3
#define ADD_BTN_CID 4
#define CHG_BTN_CID 5
#define DEL_BTN_CID 6
#define COL_BASE_CID 100
```

Control IDs only need to be unique within each interface.

4.3 XI Attributes

One parameter of the convenience functions that controls many of an object's behavior and appearance is the attribute parameter. The attribute parameter consists of one or more constants, bitwise OR'ed together. For example, the attribute parameter would be passed as `XI_ATR_ENABLED | XI_ATR_VISIBLE` to make a control visible and enabled. If you want to change the attributes of an object after it has been

created, you can get them by calling `xi_get_attrib``xi_get_attrib`, and you can set them using `xi_set_attrib``xi_set_attrib`. An object's attributes are a very important factor in determining how the object will appear on the screen, though there are many other options that are set in other ways.

XIT_ITF4.4 Interface Objects

When you instantiate an XI interface, you can allow XI to create an XVT window that will be used to hold the objects in the interface. As with all XVT windows, the look and feel of the window is determined by the look and feel of the native platform. For example, on the X platforms, each top-level window will have its own menu bar, but on Microsoft Windows, the same XVT code will create windows that are nested inside a “task window” and the windows share a menu bar. Because most of the look and feel of a window is determined by the native platform, you cannot change how the window itself will be drawn. However, there are some options you can set for a window when defining an XI interface. Some of the characteristics of an interface are described below.

4.4.1 Modal Interfaces

If you make the XI interface modal, then the interface must be dismissed before any other user interaction can take place. XI supports two modal interface modals.

With the **modal_wait** interface, supported with XVT 4.5x only, the call to **xi_create** does not return until the interface has been dismissed.

The programming model for **modal** interfaces in XI is identical to the programming model for non-modal interfaces. The call to `xi_create``xi_create` returns immediately, and the XI event handler is called as normal. The only difference is in the look and feel of the application.

The following code, from “lstdb.c”, demonstrates setting the modal flag in the interface definition:

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)form_eh, NULL,
                          "Delete Employee", 0L );
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->modal = TRUE;
```

This demonstrates the type of extensible option that we mentioned at the beginning of Chapter 3, where we first call a convenience function, then we set a field in the returned **XI_OBJ_DEF**. As we mentioned in Chapter 3, the basic information and options for an object are specified in the convenience functions. When setting optional, advanced characteristics of objects, XI uses the style above. In this case, if you left out the line where you are setting `itfdef->v.itf->modal` to **TRUE**, the default behavior would apply and the interface would not be modal.

4.4.2 Virtual Interfaces

Your application can specify that an XI interface is “virtual.” By this, we mean that the interface can be larger than the XVT window containing the interface. The user can “pan” across the interface by operating the horizontal and vertical scroll bars. In addition, as the user navigates from control to control, the interface is panned to show the control that has the focus.

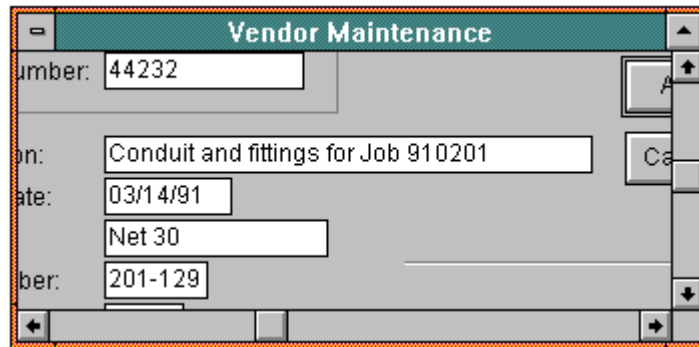


Figure 11 - A Virtual Interface

When specifying that an interface is virtual, the application must place horizontal and vertical scroll bars on the XVT window containing the interface. Failure to do this will result in internal errors when XI attempts to set the position of the elevators of the scroll bars.

An application specifies that an interface is virtual by setting the field `itf_def->v.itf->virtual_itf` to **TRUE** after calling `xi_create_itf_def`.

The following is an example of code to create an interface definition where the window has horizontal and vertical scroll bars, and the interface is virtual:

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)form_eh, NULL,
                          "Vendor Maintenance", 0L );
itfdef->v.itf->auto_back_color = TRUE;
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->ctl_hscroll = TRUE;
itfdef->v.itf->ctl_vscroll = TRUE;
itfdef->v.itf->virtual_itf = TRUE;
```

You can, of course, create your own window, and place an XI interface in the window. If you do this, it is not necessary to set `ctl_hscroll` and `ctl_vscroll` values to **TRUE**. These fields are ignored if your application creates a window before calling `xi_create`. However, if you create your own window, and if you specify that the interface is virtual, don't forget to create horizontal and vertical scroll bars on your window.

If you place XVT controls other than those supported by XI in a virtual XI interface, XI will not know about these controls, and will not move them appropriately as the user pans over the virtual interface. However, your application can respond to the `XIE_VIR_PAN` event, and move controls as necessary.

When you create a virtual interface, XI starts using a virtual coordinate system for its own internal drawing. If you need to draw other figures and graphics in an XI window that contains a virtual interface, you can use the XI drawing functions. These functions do the appropriate coordinate conversions automatically. Their prototypes are in XI.H. If you need to use other XVT drawing functions, you can determine the current delta between the physical coordinate system and the virtual coordinate system in the following fashion:

- Define `XI_INTERNAL` before including XI.H. This will allow your application to get at the internals of `XI_OBJs`.
- Use the following fields in an interface `XI_OBJ`: `xi_obj->v.itf->delta_x`, and `xi_obj->v.itf->delta_y`. Subtract these values from your x and y coordinates before drawing.

4.4.3 Putting XI Interfaces in Existing Windows

To put the interface into the existing window, set the **win** field in the **XI_ITF_DEF** structure, after calling **xi_create_itf_def**.

The following code fragment, from “lstdb.c”, demonstrates how to use this method to set different XVT flags on the window.

```
{
    RCT r;

    xi_get_def_rect( itfdef, &r );
    xvt_rect_offset( &r, (short)xi_get_pref( XI_PREF_ITF_MIN_LEFT ),
                    (short)xi_get_pref( XI_PREF_ITF_MIN_TOP ) );
    itfdef->v.itf->win = xvt_win_create( W_DOC, &r, "Employee List",
                                        MENU_BAR_RID, TASK_WIN,
                                        WSF_SIZE | WSF_CLOSE
                                        | WSF_ICONIZABLE, EM_ALL,
                                        (EVENT_HANDLER)xi_event, 0L );
}
xi_create( NULL, itfdef );
```

When you create your own window, the values in the interface definition that are used to create the window are ignored. These members are **title**, **ctl_size**, **ctl_vscroll**, **ctl_hscroll**, **ctl_close**, **rectp** and **menu_bar_rid**. Also, you cannot create your own window if the interface is modal.

4.4.4 Background Color

Your application can specify that a background color be drawn automatically for a window that contains an XI interface. You specify the color by setting the **back_color** value after calling **xi_create_itf_def**.

An advantage of letting XI draw the background color is that XI can optimize the drawing of spreadsheets, and not draw the background color underneath the spreadsheet. This results in less flicker when the user scrolls the list horizontally and vertically.

The following code is an example of setting the background color for an interface.

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)form_eh, NULL,
                           "Vendor Maintenance", 0L );
itfdef->v.itf->back_color = COLOR_LTGRAY;
```

The appropriate background color can change when you go from platform to platform. For instance, the best background color on MS-Windows is light gray. The best background color for XVT/CH 2.1 is white, while the best background color for XVT/CH 3.0 is black.

You can set a **BOOLEAN** field, **automatic_back_colorautomatic_back_color**, for the interface, such that XI will pick the best background color for you based on the platform.

The following code, from “lstdb.c”, demonstrates setting **automatic_back_color** after calling **xi_create_itf_def**:

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh, NULL,
                           "Memory List", 0L );
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_MEM_LIST_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;
```

4.4.5 Menu Bars on Windows

You can specify the menu bar resource id when creating an XI interface, **or** provide a valid MENU_ITEM tree structure. You can create the menu outside of resources, by filling the MENU_ITEM structure. You associate it with the interface, after calling `xi_create_itf_def`, using the `menu` field.

To provide a menu from resources, after calling `xi_create_itf_def`, set the `menu_bar_rid` field in the object definition. The following code, from "lstmem.c", demonstrates this.

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh, NULL,
                          "Memory List", 0L );
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_MEM_LIST_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;
```

Another approach is to create the window yourself, with the menu bar exactly as you want it, then put the XI interface into the existing window.

4.4.6 Cutting and Pasting with XI

This section tells how to use the edit menu with an XI application.

For the edit menu to work (Cut, Copy, Paste, Clear/Delete), you need to set the `XI_ATR_EDITMENU` attribute for the appropriate XI objects.

You can only set `XI_ATR_EDITMENU` for objects of types `XIT_FIELD`, and `XIT_COLUMN`.

If you have an edit menu, and want XI to automatically enable and disable the menu items on the edit menu as the user operates the application, you need to set a field when you create the interface definition. The following code, from "lstmem.c", demonstrates setting the `edit_menu` field of the interface definition structure:

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh, NULL,
                          "Memory List", 0L );
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_MEM_LIST_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;
```

Be sure that the tags for the edit menu items are the standard ones that are defined in XVT. If you use a resource menu, it's best to use the default edit menu. If you are creating your own, be sure to set the tags correctly. Refer to XVT documentation for more information about menus.

4.4.7 Scroll Bars

The window containing the interface can have a horizontal or vertical scroll bar (or both). If you choose to have either, the window will be created with them, and as the user operates the scroll bars, XVT scroll bar events are generated, and in turn, the XI events, `XIE_XVT_EVENT` and `XIE_XVT_POST_EVENT` are generated. See the XVT documentation for information about scroll bars in a window and the events they generate.

If you create the XI interface as a virtual interface, you must create the window with a horizontal and vertical scroll bar. If the XI interface is virtual, you probably will not wish to process the events generated by the scroll bars on the window. Instead, let XI process the events.

The following code is an example of putting both horizontal and vertical scroll bars on a window that contains a virtual XI interface.

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)form_oh, NULL,
                           "Vendor Maintenance", 0L );
itfdef->v.itf->auto_back_color = TRUE;
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->ctl_hscroll = TRUE;
itfdef->v.itf->ctl_vscroll = TRUE;
itfdef->v.itf->virtual_itf = TRUE;
```

4.4.8 Close Box

If the window containing the interface has a close box, the user can close the window by clicking on the close box. Otherwise, the user cannot directly close the window.

The following code, from “lstdb.c”, is an example of putting a close box on a window that contains an XI interface.

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)form_oh, NULL,
                           "Vendor Maintenance", 0L );
itfdef->v.itf->ctl_close = TRUE;
```

4.4.9 Size Controls

Sizing controls allow the interface window to be resized and maximized.

The following code, from “lstdb.c”, is an example of putting a size control on a window that contains an XI interface.

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_oh, NULL,
                           "Memory List", 0L );
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_MEM_LIST_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;
```

4.4.10 Iconize Controls

Icon controls allow the interface window to be iconized. XI allows you to create the window, with the **iconizable** control, or with the iconizable control and initially **iconized**. In addition, on Windows, NT and OS2, you can specify the **icon_rid** used to represent the iconized window.

The following code is an example of creating a window with a iconize control.

```
XI_OBJ_DEF* itfdef;

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_oh, NULL,
                           "Memory List", 0L );
itfdef->v.itf->iconizable = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_MEM_LIST_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;
```


4.4.11 Border Style

XI allows you to specify the type of interface window to create, with **border_style** and **border_style_set**. To get the style specified with **border_style** you must also set **border_style_set** to TRUE.

XinBorderSizable: Is a document window, which has a size control.

XinBorderSingle: Is a plain window that does not have a close box.

XinBorderDouble: Is a double border window that does not have a close box.

XinBorderNone: Is a window that does not have a border or close box.

XinBorderFixed: Is a document window.

The border style takes precedence over the controls listed above.

The following code is an example of creating a W_DOC window with a iconize control.

```
XI_OBJ_DEF* itfdef;  
  
itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh, NULL,  
                           "Memory List", 0L );  
itfdef->v.itf->menu_bar_rid = MENU_MEM_LIST_RID;  
itfdef->v.itf->automatic_back_color = TRUE;  
itfdef->v.itf->edit_menu = TRUE;  
itfdef->v.itf->iconizable = TRUE;  
itfdef->v.itf->border_style = XinBorderFixed;
```

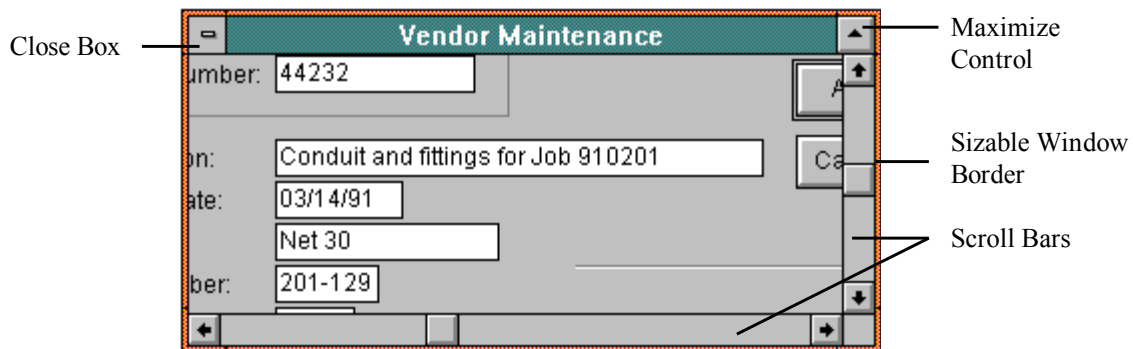


Figure 12 - XI Interface with close box, scroll bars and sizing

XIT_LIST4.5 List Objects

There are many things you might want to do with a spreadsheet list. You need to define columns and rows. You may want to label the columns with a heading, disable some columns or rows, center or right justify text in cells and so on. There are lots and lots of options for the appearance and behavior of a list. Some of these options are described in the section about *Columns*. In the following sections, you will find an overview of the options you can set for a list as a whole.

4.5.1 Disabled lists

In XI, lists can be disabled or invisible. In either case, the user cannot click or tab onto them, and therefore cells in the list cannot gain the focus. The user can tell a list is disabled because it will not accept the input focus. In addition, the application can set the colors of a list such that it looks different when disabled.

You can create a disabled list by not setting the `XI_ATR_ENABLED` attribute when calling `xi_add_list_def`.

4.5.2 Enabled lists

When a list is enabled and visible, users can move the focus into a cell in the list in one of three ways. They can either click on the cell with the mouse, tab or backtab onto it, or use the metatab key to return to it. If the focus had left the list, your event handler will receive an `XIE_ON_LIST` event when the list gains the focus, followed by `XIE_ON_ROW` and `XIE_ON_CELL` events.

You can create an enabled list by setting the `XI_ATR_ENABLED` attribute when calling `xi_add_list_def`.

4.5.3 No Column Headings

You may wish to suppress the column headings for a list. The following code, from “lstcol.c”, demonstrates setting the `no_heading` flag on a list definition.

```
list_def = xi_add_list_def( itf_def, LIST_CID, 0, 0, 8 *
XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          ADD_BTN_CID );
list_def->v.list->scroll_bar = TRUE;
list_def->v.list->no_heading = TRUE;
```

4.5.4 Horizontal Scrolling

Your application can make a horizontally scrolling list. When you have a horizontally scrolling list, XI places a horizontal scroll bar at the bottom of the list.

Users (or your application) can scroll the list horizontally by three methods:

- The users can navigate from cell to cell. XI always makes the cell with the focus visible.
- The users can operate the horizontal scroll bar: They can click the left or right arrows. They can click the left or right page areas. This will cause the list to scroll horizontally by several columns. They can also drag and drop the horizontal scroll bar thumb.
- Your application can cause the list to scroll by calling `xi_move_focus` or `xi_set_focus`, moving or setting the focus to a cell object. XI will always make the cell with the focus visible.

Your application makes a horizontal scrolling list by specifying a width for the list. This width becomes the width of the list, and columns are scrolled within that width. Your application sets the width by setting `list_def->v.list->width` after calling `xi_add_column_def`. The width of the list can be changed by calling `xi_set_list_size`. The width may also change if you set `resize_with_window` to `TRUE`. In that case, the list will be resized whenever the window resizes so that the bottom-right corner of the list matches the bottom-right corner of the window. Do not use this option if there are any other controls below or to the right of the list.

If your application does not set the **width** field, then the list will not be a horizontally scrolling list, and the list width will be the sum of the width of all of the columns.

It may be desirable to have one or more columns “fixed” at the left side of the list. These may be “title” columns. Your application can specify the number of fixed columns by setting **list_def->v.list->fixed_columns**. If this field is set, the horizontal scroll bar will not be placed under these columns, and only the columns to the right of **fixed_columns** will scroll horizontally.

The following code, from “lstmem.c”, is an example of creating a horizontally scrolling list with one fixed column.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );

listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

4.5.5 Movable Columns

Your application can set an option such that the user can “pick up” a column heading and drop it between two other column headings. When the user does this, the columns are reordered. The column is placed between the columns where the column border is closest to the hot point of the mouse cursor. Columns may even be moved from the “fixed” to the “scrolling” portions of horizontally scrolling lists with fixed columns (and vice versa).

If available, the mouse cursor changes to a “hand” cursor when the mouse pointer is over the column headings. To re-order columns, first the user moves the mouse pointer over a column heading. Then the user “drags” the column heading. While dragging, an outline of the column heading indicates where the column will be placed if the mouse button is released. Finally, the user “drops” the column heading. After dropping the column heading the columns are re-ordered.

When movable columns are enabled, a double click on the column heading will select the column, if it has the **XI_ATR_COL_SELECTABLE** attribute set, unless **XI_PREF_SINGLE_CLICK_COL_SELECT** is set to TRUE.

To enable this feature, set **list_def->v.list->movable_columns** to **TRUE** after creating the list definition by calling **xi_add_list_def**. If you wish to disable movable columns for particular columns, respond to the **XIE_COL_MOVE** event. The following code, from “lstmem.c”, is an example of creating a horizontally scrolling list with movable columns.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );

listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
```

```
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

Your application can also programmatically move a column by calling `xi_move_column`. See `xi_move_column` in the *XI Programmer's Reference* for further details.

Your application can get the layout of a list, and save it to a disk file. In this fashion, your users can modify the layout of a list, and save the configuration of the list. Then, the next time that they start their application, the list will be in the same state as it was when they last quit the application. See `xi_get_def` for further details.

4.5.6 Resizing Columns

You can allow users to resize columns. This feature can be enabled on a list-by-list basis. In addition, to selectively disable this feature, an application can refuse the `XIE_COL_SIZE` event on a column-by-column basis.

Users resize columns by dragging the border between column headings. In other words, if users click on the column headings, they can select or move columns. If users drag the border *between* the column headings, they can resize the column.

To allow users to resize columns, set the `sizable_columns` field in the `XI_LIST_DEF` structure after calling `xi_add_list_def`. The following code, from “`lstmem.c`”, is an example of creating a horizontally scrolling list with sizable column.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );

listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

4.5.7 Dynamically Deleting Columns

Your application can set an option such that when the user drags a column heading and drops it off of the list, the column is deleted. Your application enables this feature by setting the field `list_def->v.list->drop_and_deletedrop_and_delete` after calling `xi_add_list_def`. The list must also have movable columns in order to be able to drag the column heading. Typically, the list will also have a scroll bar button that will open a list that contains the columns that can be added to the list. The “Memory” list in the example program demonstrates this. The following code, from “`lstmem.c`”, is an example of a list that has movable columns and allows columns to be dynamically deleted.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );

listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
```

```
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

4.5.8 Positioning and Inserting Columns in a List

You can specify the exact position to insert a column in a list by setting the **position** field of the list definition before calling **xi_create**. For instance, if you want the column to be inserted at the left edge of the list, specify a position of 0. If you want the column to be inserted to the right of the left-most column, specify a position of 1. If you want the column to be placed to the right of all existing columns, specify a position of **SHRT_MAX**. This is only an issue if you are adding a new column to an already existing list object.

The **sort_number** field is made obsolete by this feature, but has been maintained to support existing programs.

4.5.9 List Button

The list button, or scroll bar button, is often used to open any kind of configuration dialog for a list. In the “Memory” list example, it is used to bring up a list of deleted columns that can then be added back into the list. When the user presses this button, XI generates an **XIE_BUTTONXIE_BUTTON** event with **xiev->v.xi_obj** set to the list object.

You create a button at the top of the vertical scroll bar by setting **list_def->v.list->scroll_bar_button** to

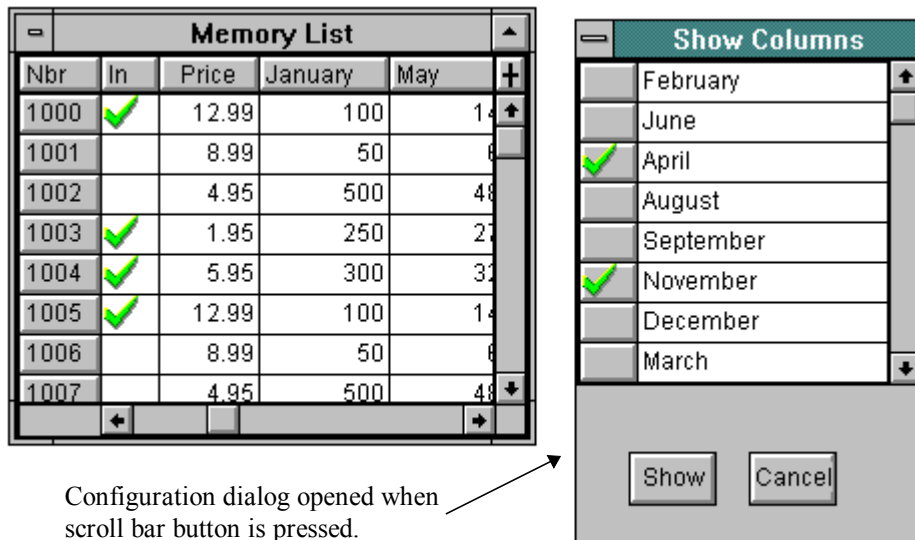


Figure 13 - List With a Scroll Bar Button

TRUE after calling **xi_add_list_def**. The following code, from “lstmem.c”, is an example of creating a list with a button.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );
listdef->v.list->scroll_bar = TRUE;
```

```

listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;

```

4.5.10 Removing Horizontal and Vertical Rules of a List

For some applications, it may be desirable to remove the horizontal or vertical rules of a list. When your application removes the horizontal and vertical rules, the operation of the list is exactly the same. You remove the horizontal and vertical rules by setting `list_def->v.list->no_horz_lines` and `list_def->v.list->no_vert_lines` to `TRUE`.

The following code is an example of removing the horizontal and vertical rules:

```

listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );
listdef->v.list->scroll_bar = TRUE;
listdef->v.list->no_horz_lines = TRUE;
listdef->v.list->no_vert_lines = TRUE;

```

4.5.11 Resizing the List when the Window is Resized

You may want to resize a list whenever the window that contains the list is resized. When the user maximizes the window, you may want to make the list be as large as possible, so that the user can see as much data as possible. When the user makes the window smaller, you can make the list fit inside of the window. Every list has a minimum size, so you may want to limit the minimum size of the window based on the resizing list. You make a list be resized with the window by setting `list_def->v.list->resize_with_window` to `TRUE`.

Note: This technique should not be used with virtual interfaces, as resizing the window is already processed by other portions of XI.

The following code, from “lstmem.c”, is an example of making a list resize with the window.

```

listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );
listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;

```

4.5.12 Changing the Number of Fixed Columns

Occasionally, you may want to change the number of fixed columns on a horizontally scrolling list. This is done by calling `xi_set_fixed_columnsxi_set_fixed_columns` for the list object.

4.5.13 List Mouse Cursors

There are three mouse cursors that are sometimes used to manipulate an XI list. The “hand” cursor is used for moving columns and will appear in the column heading when the list has that option enabled. The “horizontal resize” cursor is used for resizing columns and will appear near the line between two column headings when the list has that option enabled. The “vertical resize” cursor is used for resizing rows and will appear near the line between two rows when a column has that option enabled (see below for more information about column objects).

The resource IDs for these cursors are set as preferences. By default, the IDs are those defined in “xi.h”. The preferences and their defaults are shown in the table below.

Preference	Define for Resource ID
<code>XI_PREF_HAND_CURSOR_RID</code>	<code>XI_CURSOR_HAND</code>
<code>XI_PREF_SIZE_CURSOR_RID</code>	<code>XI_CURSOR_RESIZE</code>
<code>XI_PREF_VSIZE_CURSOR_RID</code>	<code>XI_CURSOR_VRESIZE</code>

Table 1 - Default List Cursor IDs

4.5.14 Tabwrap Navigation

You have two options for focus navigation in a list. If the attribute `XI_ATR_TABWRAP` is set, then the focus will move to the beginning of the next row if the user presses the tab key while the focus is in the last cell of the row. Without this attribute, the focus will stay on the same row, but move to the first cell. The same rules apply, in reverse, for back-tabbing.

`XI_ATR_TABWRAP` is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_list_def`. The following code, from “lstm.c”, shows the use of this attribute.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                           XI_ATR_ENABLED | XI_ATR_VISIBLE |
                           XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                           COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                           LIST_CID );
```

4.5.15 Arrow Key Navigation

If the attribute, `XI_ATR_NAVIGATEXI_ATR_NAVIGATE` is set, the arrow keys will move the focus to the next and previous cell in much the same way as do the tab and backtab keys. Within the cell, the insertion point is moved to the right and left by typing characters, and by pressing the backspace and delete keys. If this attribute is not set then the right and left arrow keys move the insertion point inside the cell. In this case, the control-right and control-left arrow keys move the focus between cells.

`XI_ATR_NAVIGATE` is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_list_def`.

4.5.16 Refreshing a List

It is often necessary to update the information displayed in the list. This may mean updating the text in all or some of the cells, or it may mean bigger changes that involve changing the rows that are displayed.

The simplest way to update all information in a list is to call `xi_scroll` with `XI_SCROLL_FIRST` as the second argument. This will cause all record request events and cell request events to occur to update the entire list. However, this will also reposition the list to the first row. If you want to try to keep the list in the same position, you can call `xi_scroll_rec` with the appropriate information about the row that you want to appear at the top of the list. All other rows will be requested by calling the event handler. Of course, you must have at least one row in order to call `xi_scroll_rec`, so if the list has become empty, you should call `xi_scroll`.

The following code, from “`lstlink.c`” and “`datlink.c`”, demonstrates both of these calls.

```
static void refresh_list( XI_OBJ* list )
{
    int      count;
    long*    handles;

    handles = xi_get_list_info( list, &count );
    if ( count == 0 )
        xi_scroll( list, XI_SCROLL_FIRST );
    else
        link_scroll_rec( list, handles[0] );
}

void link_scroll_rec( XI_OBJ* list, long handle )
{
    xi_scroll_rec( list, handle, (COLOR)0, get_attribute( handle ), 0 );
}
```

Alternatively, you may only wish to update the text without generating any record request events. You can call the function `xi_cell_request` to force cell request events for all the cells in a list, column, or row. You can also force a single cell request event for a particular cell.

Note that if `XI_PREF_OPTIMIZE_CELL_REQUESTS` is `TRUE`, then the cell request events will not occur until the cell is visible.

The following code, from “`lstlink.c`”, demonstrates this function.

```
static void update_numbers( XI_OBJ* list )
{
    XI_OBJ* column = xi_get_obj( list, COL_BASE_CID + LINK_NUM );

    if ( column != NULL )
        xi_cell_request( column );
}
```

XIT_COLUMN4.6 Columns

As mentioned above, lists have columns as their children. For example, a list might have five columns of different widths. The column objects can be created when the list is instantiated. Column objects also can be created (or deleted) after the list has been instantiated. The visual effect of adding and removing columns from a list is pretty much what you’d expect. You see a column added or see it disappear.

In the following discussion, you will find an overview of options you can set for columns. Many of the options for columns affect all the cells for that column. Because cells are quite similar to edit fields, you will find that columns and edit fields have many of the same options. Many of the list options, discussed above, will affect all or some of the columns.

4.6.1 Disabled Columns

Columns in a list can be disabled by not setting the `XI_ATR_ENABLED` attribute. Users will know that a column is disabled because none of the cells in the column will accept the input focus, and it might look different than other columns in the list if the application uses a different color to reflect its disabled state.

XI_ATR_ENABLED is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_column_def`.

4.6.2 Enabled columns

When a column is enabled and visible, users can move the focus into cells in a column in one of three ways. They can click on a cell with the mouse, tab or backtab onto it, or use the metatabmetatab key to return to a cell they were editing before they left the list to do something else on the interface. Regardless of how they get there, your event handler will receive an **XIE_ON_COLUMN** event followed by an **XIE_ON_CELL** event when the cell in the column gains the focus. Columns are enabled by setting the **XI_ATR_ENABLED** attribute, which is bitwise OR'ed together with other attributes when calling `xi_add_column_def`.

The following code, from "lstm.c", demonstrates the use of attributes on a column.

```
#define STD_COL_ATR ( XI_ATR_ENABLED | XI_ATR_COL_SELECTABLE \
                    | XI_ATR_AUTOSCROLL | XI_ATR_EDITMENU )

coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_ITEM_NBR,
                           STD_COL_ATR | XI_ATR_SELECTABLE,
                           1, 6 * XI_FU_MULTIPLE, 5, "Nbr" );
```

4.6.3 Autoselected Cells in a Column

If the column has the **XI_ATR_AUTOSELECT** attribute set, then when a cell in the column gains the focus by keyboard navigation, it will be highlighted. Otherwise, only an insertion point will be displayed. If the preference **XI_PREF_AUTOSEL_ON_MOUSE** is **TRUE**, then autoselection will also occur when the user clicks on a cell that does not have focus. Clicking on a cell that has focus will position the insertion point. **XI_ATR_AUTOSELECT** is an XI attribute, and is OR'ed together with other attributes when calling `xi_add_column_def`.

4.6.4 Read-Only Columns

If a column has the **XI_ATR_READONLY** attribute set, then users cannot change the contents of the cell even though it might have the focus. **XI_ATR_READONLY** is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_column_def`.

4.6.5 Autoscroll Cells in a Column

If a column has the attribute **XI_ATR_AUTOSCROLL** set, users can type more characters than can be displayed in a cell up to the limit you set by calling `xi_set_bufsize`, which sets the maximum length of a string that the user can type. Almost all columns will have this attribute set because of the large variation between capital and lower case letters in a proportionally spaced font. **XI_ATR_AUTOSCROLL** is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_column_def`.

4.6.6 Right-justified Columns

If a column has the attribute **XI_ATR_RJUST** set, the text in the cell of the column will be displayed as right justified. In this case, the heading text will also be right justified. When typing in a cell of a right-justified column, the text stays fixed to the right side of the cell, while characters are inserted to the left. If the **XI_ATR_RJUST** attribute is not set, then the column will be left justified. In left justified cells, the insertion point moves to the right as the user types characters.

XI_ATR_RJUST is an XI attribute, and is bitwise OR'ed together with other attributes when calling **xi_add_column_def**. This attribute can also be set for individual cells by setting the **attrib** field in response to an **XIE_CELL_REQUEST** event.

4.6.7 Password Columns

If the attribute, **XI_ATR_PASSWORD** is set, you will have a password column where the text displayed as a '#' for each character in the cell. **XI_ATR_PASSWORD** is an XI attribute, and is bitwise OR'ed together with other attributes when calling **xi_add_column_def**.

4.6.8 Platform and Well Columns

In some cases, you may want an entire column to have the platform or well appearance. To give your list platform columns, set **column_def->v.column->column_platform** to **TRUE** after calling **xi_add_column_def**. To give your list well columns, set **column_def->v.column->column_well** to **TRUE** after calling **xi_add_column_def**.

Memory List						
Nbr	Description	In	Price	January	February	Mar
1000	Widget	✓	12.99	100	110	
1001	Gadget		8.99	50	54	
1002	Socket		4.95	500	495	
1003	Bauble	✓	1.95	250	257	
1004	Sprocket	✓	5.95	300	305	
1005	Widget 2	✓	12.99	100	110	
1006	Gadget 2		8.99	50	54	
1007	Socket 2		4.95	500	495	

Figure 14 - A Platform Column

The following code, from "lstmem.c", is an example of making a platform column.

```
coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_ITEM_NBR,
                           STD_COL_ATR | XI_ATR_SELECTABLE,
                           1, 6 * XI_FU_MULTIPLE, 5, "Nbr" );

#if THREE_DIMENSIONAL
coldef->v.column->heading_platform = TRUE;
coldef->v.column->column_platform = TRUE;
#endif
coldef->v.column->size_rows = TRUE;
```

4.6.9 Centered Column Headings

For certain columns, you may prefer to have the text in the column heading be centered instead of left or right justified. To center a heading for a column, set **column_def->v.column->center_heading** to **TRUE** after calling **xi_add_column_def**.

The following code, from "lstmem.c", is an example of centering a heading over a column.

```

coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_DESCRIPTION,
                           STD_COL_ATR, 2, 20 * XI_FU_MULTIPLE,
                           MAX_DESCR, "Description" );
#if THREE_DIMENSIONAL != 0
coldef->v.column->heading_platform = TRUE;
#endif
coldef->v.column->center_heading = TRUE;

```

4.6.10 Fonts for Column Headings

Your application can set the font for a column heading. To do this, set the **column_def->v.column->font_id** field. You will need to create the font using XVT functions. The font is copied by XI during the **xi_create** function and can be destroyed after that function returns.

The following code, from “lstm.c”, demonstrates setting the font for a column heading.

```

extern XVT_FNTID xi_sysfont;
...
coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_PRICE,
                           STD_COL_ATR, 4, 12 * XI_FU_MULTIPLE,
                           7, "Price" );
#if THREE_DIMENSIONAL != 0
coldef->v.column->heading_platform = TRUE;
#endif
coldef->v.column->center_heading = TRUE;
coldef->v.column->font_id = col_font_id = xvt_font_create();
xvt_font_copy( col_font_id, xi_sysfont, XVT_FA_ALL );
xvt_font_set_style( col_font_id, XVT_FS_BOLD );
...
itf = xi_create( NULL, itfdef );
xvt_font_destroy( col_font_id );

```

If the column definition was returned from a call to **xi_get_def**, then the **font** field will also be set. This is done for compatibility with XVT R3 programs. You will need to set this field to **NULL** if you want to change the **font_id** field. Also, you should still create and destroy your own font if you want to change that **font_id**. XI will still destroy its font when the column is deleted.

4.6.11 Icons in Column Headings

You may put an icon in the column heading in place of text, or along with it. To put an icon in a column heading, set the **icon_rid** field in the column definition, after calling **xi_add_column_def**. You will also need to set **icon_mode**. You may wish to offset the icon from the upper left corner of the column heading. To do this, set the **icon_x**, and **icon_y** fields in the column definition, after calling **xi_add_column_def**.

If your icons are taller than the default heading height, you may wish to set **min_heading_height** for the list. See the section on *Multiline Column Headings* for more details.

The following code, from “lstm.c”, demonstrates setting the icon resource ID after adding the column definition to a list definition.

```

coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_IN_STOCK,
                           STD_COL_ATR | XI_ATR_SELECTABLE,
                           3, 4 * XI_FU_MULTIPLE, 2, "In" );
#if THREE_DIMENSIONAL != 0
coldef->v.column->heading_platform = TRUE;
#endif
coldef->v.column->icon_rid = ICON_CHECK;

```

4.6.12 Multiline Column Headings

There are two pertinent points about making multiple line column headings:

1. You must increase the minimum heading height to make enough room for the multiple lines.
2. XI looks for a newline ('\n') to mark the break between lines.

The following code, from "lstlink.c", demonstrates forcing a minimum pixel height for a column heading.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 3 * XI_FU_MULTIPLE, 0,
                          8 * XI_FU_MULTIPLE, XI_ATR_ENABLED
                          | XI_ATR_VISIBLE | XI_ATR_TABWRAP,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          COLOR_WHITE, COLOR_BLACK, CONTAINER_CID );
listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->width = 50 * XI_FU_MULTIPLE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->min_heading_height = 32;
```

The "lstlink.c" example uses a data structure to create its columns. The text for those columns appears in this structure and shows the embedded newlines.

```
static struct _s_coldefs
{
    LINK_FIELD field;
    short width;
    short bufsize;
    BOOLEAN rjust;
    char *title;
} coldefs[] =
{
    { LINK_NUM,      8 * XI_FU_MULTIPLE,  5,          TRUE, "Number" },
    { LINK_DATE,    11 * XI_FU_MULTIPLE, DATE_LEN + 1, FALSE, "Date" },
    { LINK_DESCR,   30 * XI_FU_MULTIPLE, 400, FALSE, "Description" },
    { LINK_WHO,     12 * XI_FU_MULTIPLE, WHO_LEN + 1,  FALSE, "Who" },
    { LINK_EST_HRS, 10 * XI_FU_MULTIPLE,  5,          TRUE,
"Estimated\nHours" },
    { LINK_ACT_HRS, 10 * XI_FU_MULTIPLE,  5,          TRUE, "Actual\nHours" },
    { 0, 0 }
};
```

Linked List					
Add All Recs		Add One Rec		Delete All Recs	
Delete Current Rec		Delete Selected Recs			
Number	Date	Description	Who	Estimated Hours	Actual Hours
1	01/01/95	This is description #1 which is long enough for word wrap.	Person 1	2	0
2	01/01/95	Alternate description #1	Person 2	4	0
3	01/01/95	Miscellaneous description #1	Person 3	6	0

Figure 15 - Multiline Column Headings

4.6.13 Platform and Well Headings

Throughout the examples, we use a “platform” appearance for column headings. To give your list platform headings, set `column_def->v.column->heading_platform` to `TRUE` after calling `xi_add_column_def`. You can also make the columns indented by setting `column_def->v.column->heading_well` to `TRUE` after calling `xi_add_column_def`.

The following code, from “lstmem.c”, is an example of making platform headings.

```
coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_DESCRIPTION,
                           STD_COL_ATR, 2, 20 * XI_FU_MULTIPLE,
                           MAX_DESCR, "Description" );
#if THREE_DIMENSIONAL != 0
coldef->v.column->heading_platform = TRUE;
#endif
coldef->v.column->center_heading = TRUE;
```

4.7 Cells and Rows

Most of the options described in this chapter are set on the object definitions. However, since rows are supplied by the application through the record request events (`XIE_GET_FIRSTXIE_GET_FIRST`, `XIE_GET_NEXTXIE_GET_NEXT`, `XIE_GET_PREVXIE_GET_PREV` and `XIE_GET_LASTXIE_GET_LAST`), the options for rows must be set there. Also, cells are supplied by the application through the `XIE_CELL_REQUESTXIE_CELL_REQUEST` event, so the options for cells are set there. Some of these options can be changed later by creating row or cell “pseudo-objects” and then calling the appropriate XI function with that object. However, cells can usually be updated by calling `xi_cell_request` and responding to the cell request event with the new options.

4.7.1 Selected Rows and Enabled Rows

A row will be selected if its attribute includes `XI_ATR_SELECTED`. Selected rows have an “inverted” color appearance. A row will be enabled if its attribute includes `XI_ATR_ENABLED`. An enabled row allow editing or focus in cells for the row. A disabled row does not allow editing or focus in the cells for that row and may have a different color if disabled colors are specified. These attributes can be bitwise OR’ed together and set in the `attrib` field of the record request event structure.

The following code, from “datmem.c”, demonstrates setting the “selected” attribute.

```

void mem_rec_request( REC_INFO* rec, XI_EVENT* xiev )
{
    ...
    if ( rec->row_selected )
        xiev->v.rec_request.attrib |= XI_ATR_SELECTED;
}

```

Memory List							
Nbr	Description	✓	Price	January	February	March	April
1000	Widget	✓	12.99	100	110	120	
1001	Gadget		8.99	50	54	58	
1002	Socket		4.95	500	495	490	
1003	Bauble	✓	1.95	250	257	264	
1004	Sprocket	✓	5.95	300	305	310	
1005	Widget 2	✓	12.99	100	110	120	
1006	Gadget 2		8.99	50	54	58	
1007	Socket 2		4.95	500	495	490	

Figure 16 - A Selected Row

4.7.2 Colors Per Cell

Your application can set both the foreground and background colors for cells. This is done by setting the **color** and **back_color** fields in the cell request event structure. If you want to update the colors for a cell or cells, you should call **xi_cell_request** and respond to the resulting **XIE_CELL_REQUEST** events with the new colors.

The following code, from “datmem.c”, demonstrates setting colors for cells.

```

void mem_cell_request( REC_INFO* rec, VALUE_CODE code, XI_EVENT* xiev )
{
    ...
    xiev->v.cell_request.attrib = rec->attrib[ (int)code ];
    if ( rec->selected[ (int)code ] )
        xiev->v.cell_request.attrib |= XI_ATR_SELECTED;
    if ( rec->have_font[ (int)code ] )
        xiev->v.cell_request.font_id = rec->font_ids[ (int)code ];
    xiev->v.cell_request.color = rec->colors[ (int)code ];
    xiev->v.cell_request.back_color = rec->back_colors[ (int)code ];
}

```

4.7.3 Fonts Per Cell

Your application can set a font for the text in a cell. This is done by setting the **font_id** field of the cell request event structure. If you want to update the font for a cell or cells, you should call **xi_cell_request** and respond to the resulting **XIE_CELL_REQUEST** events with the new font.

The following code, from “datmem.c”, demonstrates setting fonts for cells.

```

void mem_cell_request( REC_INFO* rec, VALUE_CODE code, XI_EVENT* xiev )
{
    ...
    xiev->v.cell_request.attrib = rec->attrib[ (int)code ];
    if ( rec->selected[ (int)code ] )
        xiev->v.cell_request.attrib |= XI_ATR_SELECTED;
    if ( rec->have_font[ (int)code ] )
        xiev->v.cell_request.font_id = rec->font_ids[ (int)code ];
    xiev->v.cell_request.color = rec->colors[ (int)code ];
    xiev->v.cell_request.back_color = rec->back_colors[ (int)code ];
}

```

IMPORTANT: Although XI copies the font after you return from the cell request event, you are responsible for destroying any of the XVT fonts that you create. Since this can't be done after returning from the event, you will probably need to keep track of these fonts and then destroy them when the interface is deleted. The following code, from "datmem.c", is called from the **XIE_CLEANUP** event for the interface (that code is in "lstmem.c").

```

void mem_free_fonts( REC_INFO* rec )
{
    int num;

    for ( num = 0; num < MAX_COLUMNS; num++ )
        if ( rec->have_font[ num ] )
            xvt_font_destroy( rec->font_ids[ num ] );
}

```

4.7.4 Cell Range Selection

You may wish to allow your users to select a range of cells. After they have selected a range of cells, you could allow them to change the font of the cells, or colors of cells. The "Memory" list demonstrates the use of selected cells for this purpose. This option is enabled by setting the **select_cells** field in the list definition structure.

Memory List							
Nbr	Description	Price	January	February	March	April	
1000	Widget	12.99	100	110	120		
1001	Gadget	8.99	50	54	58		
1002	Socket	4.95	500	495	490		
1003	Bauble	1.95	250	257	264		
1004	Sprocket	5.95	300	305	310		
1005	Widget 2	12.99	100	110	120		
1006	Gadget 2	8.99	50	54	58		
1007	Socket 2	4.95	500	495	490		

Figure 17 - A Range of Selected Cells

There are several things to know about allowing users to select cells:

- Users select a range of cells by placing the mouse cursor over an intersection of cells. On some platforms, when the mouse is in position to select a range of cells, the cursor changes to a "plus".
- Your application can get or set the **XI_ATR_SELECTED** attribute for cells using the **xi_get_attrib** and **xi_set_attrib** functions. To set or get an attribute this way, you need to fabricate a cell object.

- When XI sends an **XIE_CELL_REQUEST** event, you can indicate that the cell is selected by setting the **attrib** field in the cell request event structure.
- When the user selects a range of cells, you will get an event, **XIE_SELECT**, with **xiev->v.xi_obj** set to the list object. If cells were previously selected, they will be deselected when this event occurs.
- Your application can retrieve the selected range of cells by calling **xi_get_cell_selection**. See **xi_get_cell_selection** in the *XI Programmer's Reference* for further details.

Currently, XI does not allow automatic scrolling while selecting a range of cells. This is because of the added difficulty of holding more record handles than are visible while those cells are selected. We plan to add this feature in a future version.

The following code, from "lstmem.c", shows how to enable cell range selection.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );

listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

The following code, from "lstmem.c", demonstrates responding to the **XIE_SELECT** event.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

    switch ( xiev->type )
    {
        ...
        case XIE_SELECT:
            switch ( xiev->v.select.xi_obj->type )
            {
                case XIT_ROW:
                {
                    ...
                    break;
                }
            }
    }
}
```



```

case XIT_LIST:
{
    XI_CELL_SPEC* cells;
    int count;
    int num;
    XI_OBJ* list = xiev->v.select.xi_obj;

    cells = xi_get_cell_selection( list, &count );
    select_clear( list->itf, FALSE, FALSE );
    for ( num = 0; num < count; num++, cells++ )
        mem_select_cell( row_to_record( list, cells->row ),
                        column_to_code( list, cells->column ),
                        xiev->v.select.selected );

    break;
}
}
break;

```

The function **select_clear** appears earlier in “lstmem.c”. The **mem_select_cell** function appears in “datmem.c” as follows:

```

void mem_select_cell( REC_INFO* rec, VALUE_CODE code, BOOLEAN flag )
{
    rec->selected[ (int)code ] = flag;
}

```

The following code, from “datmem.c”, shows how the “selected” attribute is set in the cell request structure.

```

void mem_cell_request( REC_INFO* rec, VALUE_CODE code, XI_EVENT* xiev )
{
    ...
    xiev->v.cell_request.attrib = rec->attrib[ (int)code ];
    if ( rec->selected[ (int)code ] )
        xiev->v.cell_request.attrib |= XI_ATR_SELECTED;
    if ( rec->have_font[ (int)code ] )
        xiev->v.cell_request.font_id = rec->font_ids[ (int)code ];
    xiev->v.cell_request.color = rec->colors[ (int)code ];
    xiev->v.cell_request.back_color = rec->back_colors[ (int)code ];
}

```

If you want to handle cell range selection, you should look closely at “lstmem.c” and “datmem.c” for the techniques involved.

4.7.5 Putting Icons in Cells

You may want to put icons in cells in an XI list. Icons must be in the resource for the application. See the XVT documentation for details about using icons. To make an icon appear in a cell, you set the **icon_rid** field of the cell request structure. If you want to change the icon later, you should call **xi_cell_request** and then respond to the resulting **XIE_CELL_REQUEST** event(s) with the new icon resource ID.

The following code demonstrates setting the icon for a cell.

```

void mem_cell_request( REC_INFO* rec, VALUE_CODE code, XI_EVENT* xiev )
{
    char  buffer[ 20 ];
    char* data_ptr = buffer;

    switch ( code )
    {
        ...
        case VALUE_IN_STOCK:
            strcpy( buffer, rec->in_stock ? "T" : "F" );
            xiev->v.cell_request.icon_rid = rec->in_stock ? ICON_CHECK
                : ICON_EMPTY;
            break;
        ...
    }
}

```

On some platforms, the default height of a cell may be too short for the icon which will result in the icon being clipped at the bottom of the cell. You can force the height of all rows to be larger by setting **list_def->v.list->min_cell_height** to the desired row height in pixels. (This value is in pixels, not form units.)

4.7.6 Putting Bitmaps in Cells

You may want to put bitmaps in cells in an XI list. In the “Memory” list example, we use a check mark bitmap to indicate if an item is in stock. Bitmaps must be created in the application and kept as long as they are used. To make a bitmap appear in a cell, you set the **bitmap** field of the cell request structure. If you want to change the bitmap later, you should call **xi_cell_request** and then respond to the resulting **XIE_CELL_REQUEST** event(s) with the new bitmap pointer (created with **xi_bitmap_create**). Call **xi_bitmap_destroy** in the **XIE_CLEANUP** event, or, when you will no longer use the bitmap.

The following code, from “datmem.c”, demonstrates setting the bitmap for a cell.

```

void mem_cell_request( REC_INFO* rec, VALUE_CODE code, XI_EVENT* xiev )
{
    char  buffer[ 20 ];
    char* data_ptr = buffer;

    switch ( code )
    {
        ...
        case VALUE_IN_STOCK:
            strcpy( buffer, rec->in_stock ? "T" : "F" );
            xiev->v.cell_request.bitmap = rec->in_stock ? check_bitmap :
            empty_bitmap;
            break;
        ...
    }
}

```

On some platforms, the default height of a cell may be too short for the bitmap. You can force the height of all rows to be larger by setting **list_def->v.list->min_cell_height** to the desired row height in pixels. (This value is in pixels, not form units.)

4.8 Forms

Form objects have edit fields as their children. The form serves only three purposes. First, all edit fields must be in a form. Second, the metatab character will only tab correctly between forms, lists and containers. Third, XI generates **XIE_ON_FORM** and **XIE_OFF_FORM** events when focus enters and leaves an edit field in the form. In general, the form object plays a very small part in the operation of an XI application.

4.9 Edit Fields

If you have used XVT edit controls, you will probably notice that XI edit fields are not XVT edit controls. The XI edit fields are displayed using XVT drawing primitives. This gives them different characteristics than the XVT edit controls. For example, you can have a password edit field where your user can type characters and have a masking character displayed instead. This type of functionality doesn't come with XVT because XVT uses the native edit controls on each platform. The native edit controls are often limited in the choices of behaviors you can set for them.

An XI edit field does not include a label. You should use the “static text” object for labels for edit fields.

Below is a summary of features that XI edit fields can have. You may notice that this list is quite similar to the behaviors cells in a column can have in a list. This is because cells and edit fields share the same code internally in the XI tool kit.

4.9.1 Disabled Edit Fields

In XI, edit fields can be disabled or invisible. In either case, the user cannot click or tab onto them, and therefore they cannot gain the focus. The user can tell an edit field is disabled because it will not accept the input focus. In addition, the application can set the colors of an edit field such that it looks different when disabled. You create a disabled edit field by not setting the **XI_ATR_ENABLED** attribute when creating the edit field definition.

4.9.2 Enabled Edit Fields

When an edit field is enabled and visible, users can move the focus onto an edit field in one of three ways. They can either click on it with the mouse, tab or backtab onto it, or use the metatab key to return to it from a list, container, or other form on the same interface. Regardless of how they get there, your event handler will receive an **XIE_ON_FIELDXIE_ON_FIELD** event when the edit field gains the focus.

XI_ATR_ENABLED is an XI attribute, which is bitwise OR'ed together with other attributes when calling `xi_add_field_def`. The following code, from “lstdb.c”

4.9.3 Autoselected Edit Fields

If the edit field has the **XI_ATR_AUTOSELECT** attribute set, then the text in the edit field will be selected when it gains the focus by keyboard navigation. If the preference **XI_PREF_AUTOSEL_ON_MOUSE** is **TRUE**, then autoselection will also occur when the user clicks on a cell that does not have focus. Clicking on a cell that has focus will position the insertion point. Otherwise, only an insertion point will be displayed.

XI_ATR_AUTOSELECT is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_field_def`.

4.9.4 Read-Only Edit Fields

If an edit field has the **XI_ATR_READONLY** attribute set, then users cannot change the contents of the edit field even though it might have the focus. **XI_ATR_READONLY** is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_field_def`.

4.9.5 Autoscroll Edit Fields

If it has the attribute `XI_ATR_AUTOSCROLL` set, users can type more characters than can be displayed in the edit field up to the limit you set by calling `xi_set_bufsize`, which sets the maximum length of a string that the user can type. Almost all edit fields will have this attribute set because of the large variation between capital and lower case letters in a proportionally spaced font. `XI_ATR_AUTOSCROLL` is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_field_def`.

4.9.6 Right-justified Edit Fields

If an edit field has the attribute `XI_ATR_RJUST` set, the text in the edit field will be displayed as right justified. In a right justified edit field, characters are inserted to the left of the insertion point as the user types. If the `XI_ATR_RJUST` attribute is not set, the edit field will be left justified. In a left justified edit field, the insertion point moves to the right as the user types characters.

`XI_ATR_RJUST` is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_field_def`.

4.9.7 Password Edit Fields

If the attribute, `XI_ATR_PASSWORD` is set, you will have a password edit field where the text displayed as a '#' for each character in the edit field. `XI_ATR_PASSWORD` is an XI attribute, and is bitwise OR'ed together with other attributes when calling `xi_add_field_def`.

4.9.8 Platform and Well Edit Fields

You can create edit fields in XI that have the appearance of a well or a platform. You do this by setting `field_def->v.field->well` to `TRUE` or by setting `field_def->v.field->platform` to `TRUE`.

4.9.9 Edit Field Buttons

You can specify that XI automatically create a button to be associated with an edit field. This button can be placed to the right or left of the edit field. Typically, this button will contain an icon that has a down arrow in it, and the user will be presented with a choice of values when they presses the edit field button. An edit field has a button if the `button` member of the edit field definition structure is set to `TRUE`. That button will be on the right side of the edit field unless the `button_on_left` member of the structure is also `TRUE`.

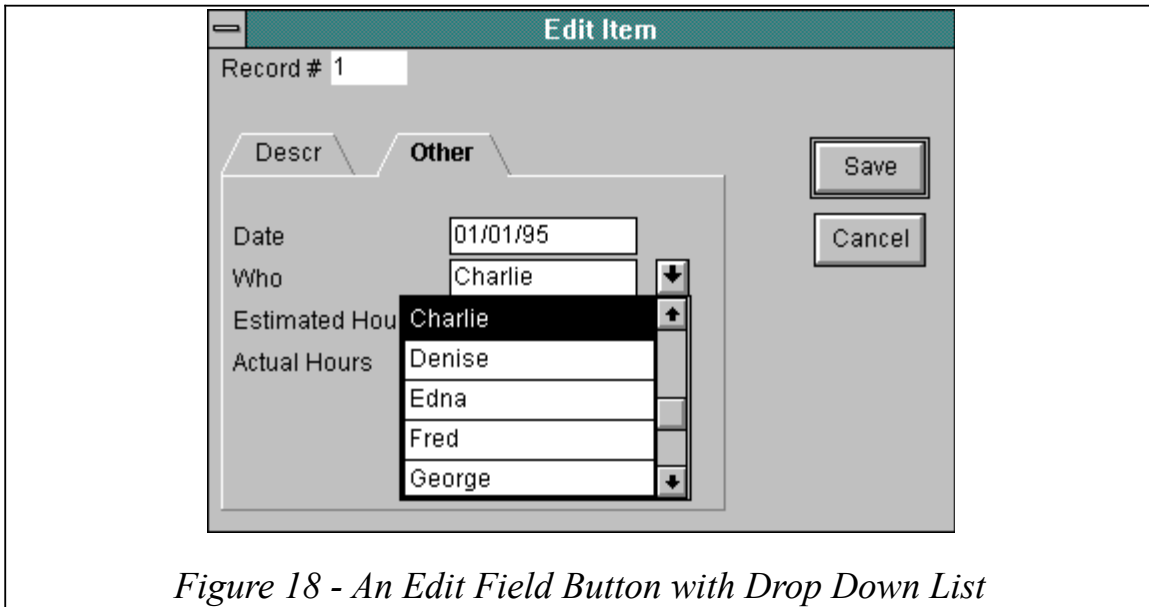


Figure 18 - An Edit Field Button with Drop Down List

When the user presses the button, two things happen. First, XI attempts to move the focus to the edit field, generating the appropriate **XIE_OFF_*** and **XIE_ON_*** events. If no event is refused, and the focus is moved to the edit field, an **XIE_BUTTON** event is generated, with **xi_ev->v.xi_obj** set to the edit field object.

Your application can specify the icon used for the edit field button by setting **field_def->v.field->icon_rid** to a resource id for an icon resource that you have created in your application resources. Refer to your XVT documentation for details on putting an icon in your resources. By default, XI uses the resource ID for **XI_PREF_COMBO_ICON** for an edit field button.

The "1stlink.c" file contains code to create an interface with a field button and a "drop down list" selection.

4.9.10 Using **XI_ATR_FOCUSBORDER**

You may wish to emphasize the location of the focus. In particular, when using the 3D look, it may be a bit hard to see the insertion point, and the user may not know where characters they type are going to go. You can set an attribute for edit fields, such that the black border around the edit fields is only drawn if the edit field has the focus. This is enabled by setting the **XI_ATR_FOCUSBORDER** for an edit field. This attribute is bitwise OR'ed with other attributes when calling **xi_add_field_def** or **xi_add_field_def**.

4.9.11 Multiline Edit Fields

This section describes making multiline edit fields. To create a multiline edit field, set the values in the **xi_rect** structure in the edit field definition structure after calling **xi_add_field_def**. If the edit field is the height of 8 form units, the edit field is multi-line. The height should never be set to less than 8 form units.

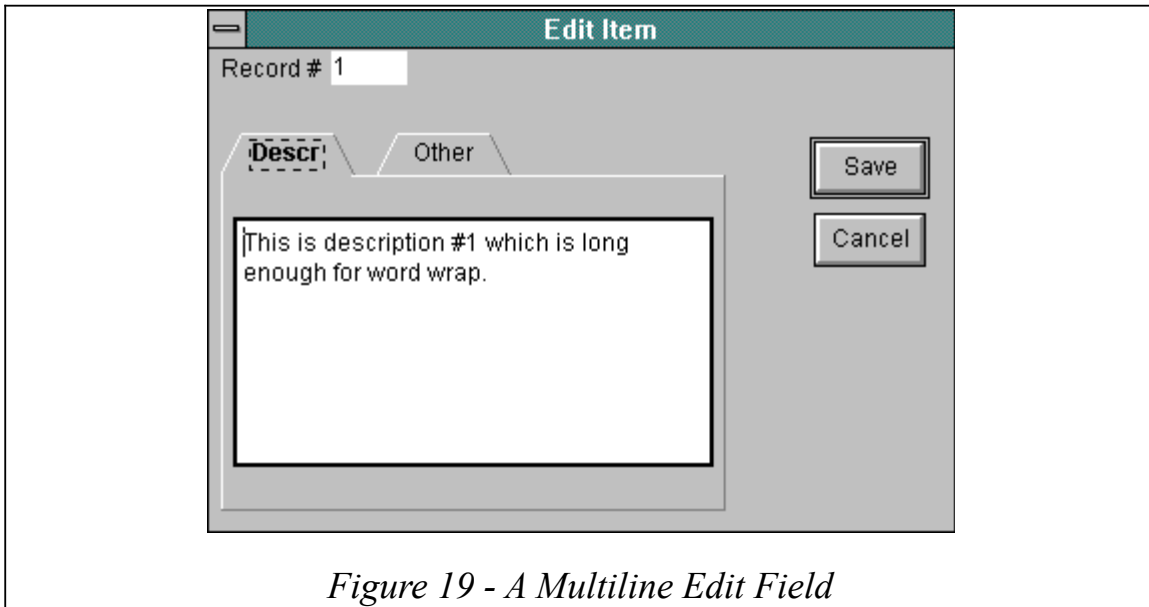


Figure 19 - A Multiline Edit Field

Because of certain characteristics of the text edit system, some **XIE_XVT_EVENT** events will be sent to the event handler before the **XIE_INIT** event is sent to the event handler.

The following code, from "lstdb.c", demonstrates setting the **xi_rct** field:

```

for ( num = 0; fielddefs[ num ].width != 0; num++ )
{
    XI_OBJ_DEF* def;
    long attrib;

    attrib = XI_ATR_AUTOSELECT | XI_ATR_AUTOSCROLL;
    if ( fielddefs[ num ].section <= 1 )
        attrib |= XI_ATR_VISIBLE;
    if ( fielddefs[ num ].enabled )
        attrib |= XI_ATR_ENABLED | XI_ATR_BORDER;
    def = xi_add_field_def( formdef, FIELD_BASE_CID
                          + fielddefs[ num ].type,
                          fielddefs[ num ].v * XI_FU_MULTIPLE,
                          fielddefs[ num ].h * XI_FU_MULTIPLE,
                          fielddefs[ num ].width * 3
                          / 2 * XI_FU_MULTIPLE,
                          attrib, ( fielddefs[ num + 1 ].type == 0 )
                          ? SAVE_BTN_CID : FIELD_BASE_CID
                          + fielddefs[ num + 1 ].type,
                          fielddefs[ num ].width + 1, COLOR_BLACK,
                          COLOR_WHITE, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK );
    if ( fielddefs[ num ].height != 0 )
    {
        XI_RCT* prct = &def->v.field->xi_rct;

        prct->top = def->v.field->pnt.v;
        prct->left = def->v.field->pnt.h;
        prct->bottom = prct->top + fielddefs[ num ].height *
        XI_FU_MULTIPLE;
        prct->right = prct->left + 40 * XI_FU_MULTIPLE;
    }
    def->v.field->button = fielddefs[ num ].button;
}

```

When you need to set the text of a multiline edit field, you can call `xi_set_text` with a multiline string. The `'\r'` character delimits each paragraph in the string. When you get the text via a call to `xi_get_text`, the `'\r'` character delimits each paragraph.

If the `cr_ok` value in the edit field definition structure is `TRUE`, the enter key is used to enter multiple lines in a multiline edit field, it cannot be used to press the default button. When the multiline edit field has the focus, the default button can only be pressed by using the mouse, or by tabbing off of the multiline edit field, then pressing enter. Also, tabs are used for keyboard navigation and cannot be entered into the text of an edit field.

4.9.12 Edit Field Fonts

Your application can have a different font for each edit field object, if you wish. To set the font for an edit field, set the `font_id` member in the edit field definition structure after calling `xi_add_field_def`. You will need to create the font using XVT functions. The font is copied by XI during the `xi_create` function and can be destroyed after that function returns.

If the edit field definition was returned from a call to `xi_get_def`, then the `font` field will also be set. This is done for compatibility with XVT R3 programs. You will need to set this field to `NULL` if you want to change the `font_id` field. Also, you should still create and destroy your own font if you want to change that `font_id`. XI will still destroy its font when the edit field is deleted.

4.10 Groups

Unlike many other XI objects, groups are abstract objects in that they have no appearance on the screen. Groups are especially useful to developers of database applications because they make it much easier to validate foreign keys that are made up of multiple fields.

A group can consist of either a group of edit fields or a group of columns. The entire purpose of the group is to generate `XIE_ON_GROUP` and `XIE_OFF_GROUP` events when focus leaves any of the objects in the group. For example, if you have a group of two edit fields, you will receive focus events for each edit field as focus moves between them. However, the `XIE_OFF_GROUP` event only occurs when you move from either of the edit fields in the group to an object that is not in the group.

The group validation has three main uses:

1. The event can trigger a record lookup for a database table that has a multiple field key.
2. The event can trigger a foreign key validation that verifies the values in another database table. Again, for the case of multiple fields in the foreign key.
3. You can validate related values. For example, in the “Employee” list, we check that the minimum hours are less than or equal to the maximum hours.

Even though groups could be thought of as having children, they do not. Every edit field in a group is actually a child of a form and every column in a group is actually a child of a list. For this reason, groups use a list of control IDs to refer to the controls that are defined elsewhere in the interface. This also allows you to put an edit field in more than one group, if you needed to do so.

4.11 Containers

Container objects have buttons as children. For this reason, they are often referred to as “button containers”. The purpose of the container object is to arrange the buttons. This is very useful when lining up buttons either horizontally, vertically or in a grid. In addition, a container is essential for radio buttons and tab buttons because it will uncheck the other buttons in the group whenever one of them is checked.

The choices for orientation are defined by the `XI_CONTAINER_ORIENTATION` enum. This value is passed to the call to `xi_add_container_def`.

The size of buttons in a container is determined by XI for horizontal and vertical orientations. The grid orientations allow you to put buttons very close together, but you cannot have a default button in that case. Use the **packed** member of the container definition structure to put gridded buttons close together. Also, when you specify a grid orientation, you can set the height and/or width of the buttons with the **btn_height** and **btn_width** members of the container definition structure.

The following code, from “lstdb.c”, shows the definition for a packed grid of icon buttons.

```
rct.top = 0;
rct.left = 0;
rct.bottom = 24;
rct.right = 144;
cntrdef = xi_add_container_def( itfdef, CONTAINER_CID, &rct,
                               XI_GRID_HORIZONTAL, LIST_CID );
cntrdef->v.container->packed = TRUE;
cntrdef->v.container->btn_width = 6 * XI_FU_MULTIPLE;

btndef = xi_add_button_def( cntrdef, ADD_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Add",
                            CHG_BTN_CID );
btndef->v.btn->down_icon_rid = ADD_BTN_ICON;
btndef->v.btn->up_icon_rid = ADD_BTN_ICON;

btndef = xi_add_button_def( cntrdef, CHG_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Chg",
                            DEL_BTN_CID );
btndef->v.btn->down_icon_rid = CHG_BTN_ICON;
btndef->v.btn->up_icon_rid = CHG_BTN_ICON;

btndef = xi_add_button_def( cntrdef, DEL_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Del",
                            ADD_BTN_CID );
btndef->v.btn->fore_color = COLOR_RED;
btndef->v.btn->down_icon_rid = DEL_BTN_ICON;
btndef->v.btn->up_icon_rid = DEL_BTN_ICON;
```

4.12 Buttons

When you instantiate an XI interface, the XI buttons you see on the screen will be either 3D buttons that are drawn by XI, or are actually XVT button controls. XI interprets the **E_CONTROL** messages sent from XVT so that XVT buttons fit into the same framework as every other XI object. Setting the preference **XI_PREF_NATIVE_CTRL**s to **TRUE** causes XI to use XVT buttons. Setting the preference to **FALSE** causes XI to draw its own 3D style buttons. To specify how a button will appear on the screen you set attributes listed below.

4.12.1 Types of XI Buttons

There are six types of XI buttons, as enumerated by **XI_BTN_TYPE****XI_BTN_TYPE**. They are:

- **XIBT_BUTTON****XIBT_BUTTON**. This is a regular button.
- **XIBT_CHECKBOX****XIBT_CHECKBOX**. This is a check box button.
- **XIBT_RADIOBTN****XIBT_RADIOBTN**. This is a radio button. When the application calls **xi_checkxi_check** on a radio button that is inside of a group, all other radio buttons in the group are unchecked.
- **XIBT_TABBTN****XIBT_TABBTN**. This is a special form of radio buttons. It operates in the same fashion as radio buttons, but it has a different appearance. (This type of button becomes a radio button if **XI_PREF_NATIVE_CTRL**s is set to **TRUE**.)

- `XIBT_BUTTON_CHECKBOX``XIBT_BUTTON_CHECKBOX`. This is a check box button with a regular button appearance. When the button is checked, it will remain “depressed”.
- `XIBT_BUTTON_RADIOBTN``XIBT_BUTTON_RADIOBTN`. This is a radio button with a regular button appearance. It operates in the same fashion as radio buttons, but the button remains “depressed” when it is checked.

4.12.2 Using XVT Buttons

We created XI buttons for several reasons. There is a problem with XVT buttons because XVT uses native controls for their buttons. When they have the focus, sometimes they intercept characters necessary for navigation. In addition, it isn’t possible to put icons, or bitmaps in XVT push buttons.

However, you can make XI use native controls by setting the preference `XI_PREF_NATIVE_CTRL``XI_PREF_NATIVE_CTRL`s to **TRUE**. (This is the default setting.) If you do this, certain features are not available, such as the ability to put icons in push buttons, the ability to have a default button in a window, and some navigation characteristics. However, using native controls may be desirable on systems that don’t have a gray scale display, such as the Macintosh classic.

On XVT/CHXVT/CH, only native buttons may be used.

4.12.3 Disabled Buttons

In XI, a button can be disabled and has a characteristic look and feel when it is. When it is disabled, the user cannot click or tab onto it, and therefore the button cannot gain the focus. Also, if the default button is disabled, the enter key will not press it.

You create a disabled button by not setting the `XI_ATR_ENABLED` attribute when creating the button definition.

4.12.4 Enabled Buttons

When a button is enabled, users can depress the button by clicking on it with the mouse, or by tabbing onto it and pressing the space bar. `XI_ATR_ENABLED` is an XI attribute, and is bitwise OR’ed together with other attributes when calling `xi_add_button_def``xi_add_button_def`.

4.12.5 Icon and Bitmap Buttons

When using XI push buttons, you have the option of putting icons or bitmaps in the push buttons. Of course, putting icons and bitmaps in buttons only works in the graphical systems, not in XVT/CH. For that reason, you should specify the text for the button so that it can be used if icons or bitmaps are not available.

When creating an icon or bitmap button, you must specify three icons or bitmaps. They are:

- The icon or bitmap displayed when the button is up.
- The icon or bitmap displayed when the button is down.
- The icon or bitmap displayed when the button is disabled.

You may want to make a certain visual relationship among these three images. For example, the up image has a 3D appearance such that it is not depressed. The down image has the same 3D appearance, but it would be depressed. The disabled image may be the same as the up image, but you might replace all colors with shades of gray to make the button appear disabled. Obviously, you have the option to simply use the same image for all three states.

When putting an icon into an icon button, you can specify the distance in pixels from the upper left where all icons are displayed. In this fashion, you can create a very small icon button, and place a small icon in the upper left corner of the button. Or you can create a very large icon button, and compute the pixel distance from the upper left such that the icon is displayed in the center of the large icon button.

Icon buttons can be created only if the value of the preference `XI_PREF_NATIVE_CTRL`s is set to `FALSE`. This is the default value.

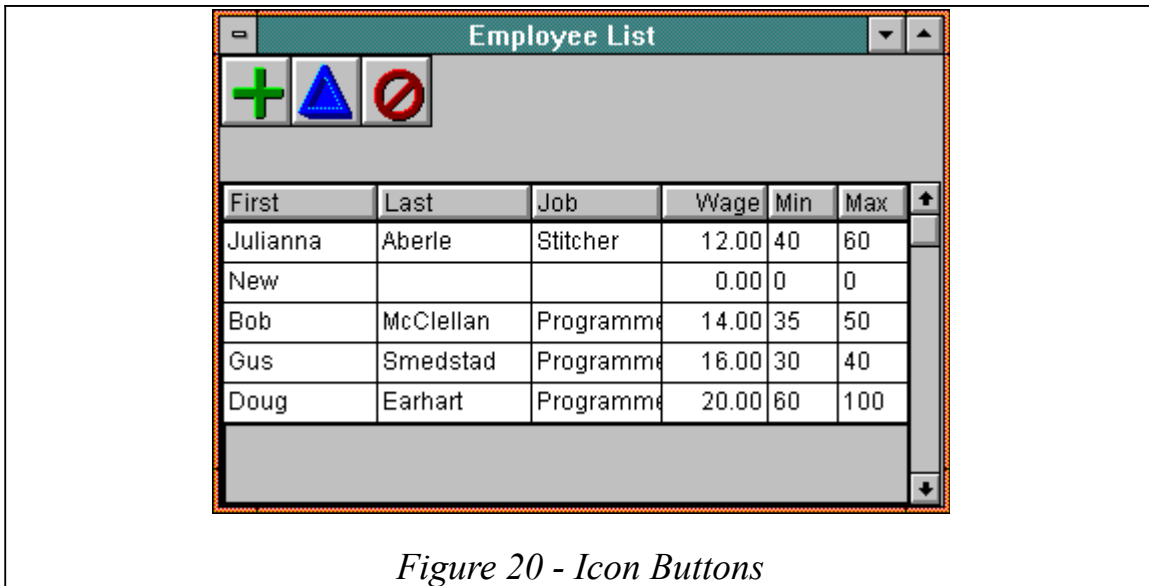


Figure 20 - Icon Buttons

The following code, from "lstdb.c", demonstrates creating icon buttons.

```

rct.top = 0;
rct.left = 0;
rct.bottom = 24;
rct.right = 144;
cntrdef = xi_add_container_def( itfdef, CONTAINER_CID, &rct,
                               XI_GRID_HORIZONTAL, LIST_CID );
cntrdef->v.container->packed = TRUE;
cntrdef->v.container->btn_width = 6 * XI_FU_MULTIPLE;

btndef = xi_add_button_def( cntrdef, ADD_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Add",
                            CHG_BTN_CID );
btndef->v.btn->down_icon_rid = ADD_BTN_ICON;
btndef->v.btn->up_icon_rid = ADD_BTN_ICON;

btndef = xi_add_button_def( cntrdef, CHG_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Chg",
                            DEL_BTN_CID );
btndef->v.btn->down_icon_rid = CHG_BTN_ICON;
btndef->v.btn->up_icon_rid = CHG_BTN_ICON;

btndef = xi_add_button_def( cntrdef, DEL_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Del",
                            ADD_BTN_CID );
btndef->v.btn->fore_color = COLOR_RED;
btndef->v.btn->down_icon_rid = DEL_BTN_ICON;
btndef->v.btn->up_icon_rid = DEL_BTN_ICON;

```

4.12.6 Radio Buttons

You create radio buttons in the same fashion that you create push buttons. However, you set **btn_def->v.btn->type** to **XIBT_RADIOBTN** after calling **xi_add_button_def**.

When creating the radio buttons, you can specify the initial state by setting **btn_def->v.btn->checked** to **TRUE** or **FALSE**. The radio button will be created with the specified initial state. Of course, you would specify that only one radio button in a set of radio buttons is initially checked.

If you elect not to use XI controls by setting the preference **XI_PREF_NATIVE_CTRL**s to **FALSE**, and instead use XVT controls, then the initial state is ignored. You will need to call **xi_check** during the **XIE_INIT** event or after the call to **xi_create** returns. Native radio buttons can only be checked if they are in a container.

There are two ways that you can create radio buttons - you can create them inside of a container, or you can create them as children of the interface. If you create the radio buttons as children of a container, when you call **xi_check** on one of the radio buttons, the other radio buttons in the container are automatically unchecked.

When making radio buttons, you can specify the foreground color. For example, you could have a certain set of radio buttons that are red, and another set that are blue.

4.12.7 Check Boxes

You create check boxes in the same fashion that you create push buttons. However, you set **btn_def->v.btn->type** to **XIBT_CHECKBOX** after calling **xi_add_button_def**.

When creating the check boxes, you can specify the initial state by setting **btn_def->v.btn->checked** to **TRUE** or **FALSE**. The check box will be created with the specified initial state.

If you elect not to use XI controls by setting the preference **XI_PREF_NATIVE_CTRL**s to **FALSE**, and instead use XVT controls, then the initial state is ignored. You will need call **xi_check** upon the **XIE_INIT** event or after the call to **xi_create** returns.

When making check boxes, you can specify the foreground color. For example, you could have a red check box and a blue check box.

4.12.8 Tab Buttons

You can make a sophisticated user interface by using tab buttons. Tab buttons look like the tabs on file folders. When the user clicks on a tab button, it visually appears to come to the front. This is the “checked” state of a tab button.

Semantically, tab buttons are identical to radio buttons. The only difference is their appearance. In other words, if your application calls **xi_check**, passing a tab button **XI_OBJ**, that tab is brought to the front, and the other tabs are “unset”, or put to the back.

Typically, your application would associate several XI objects with each tab. Then when a tab button is pressed, your application would hide any visible XI objects associated with other tab buttons, and would make visible any XI objects associated with the tab button that was just pressed. In this fashion, you can “layer” the objects in an interface so that they take up less space. The “Link List” demonstrates this in its popup dialog that appears if you double-click on a cell.

The way that your application creates a tab button is the same way that it creates radio buttons in a container. However, you set **btn_def->v.btn->type** to **XIBT_TABBTN** after calling **xi_add_button_def**.

In addition, you should create an XI rectangle such that the top border of the XI rectangle is equal to the bottom border of the container.

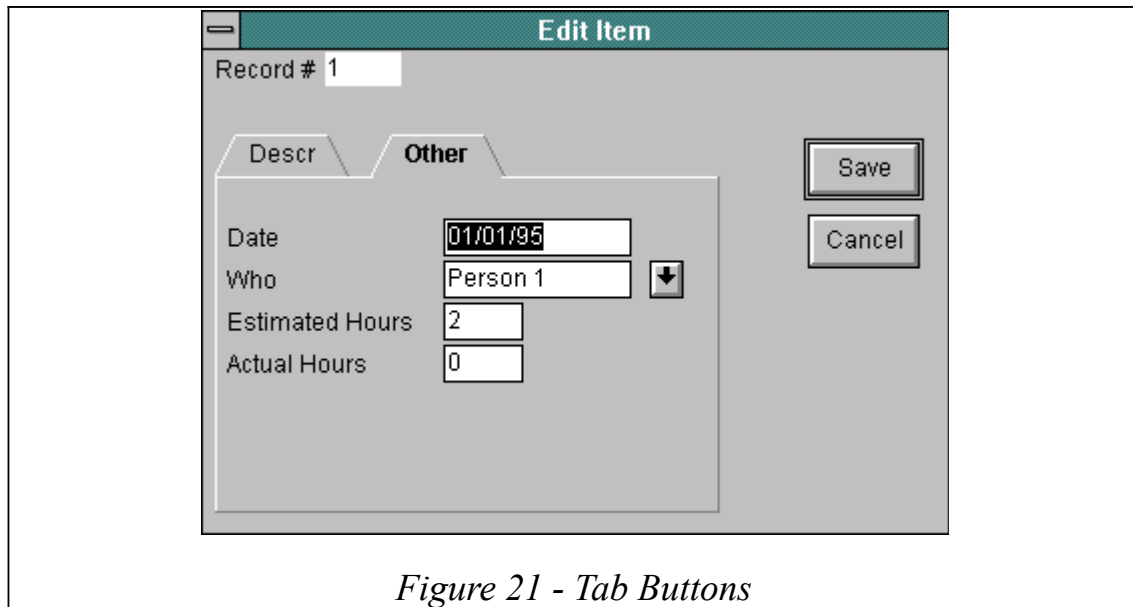


Figure 21 - Tab Buttons

The following code, from "lstlink.c", is an example of creating a container that contains tab buttons, and a rectangle that is immediately below the tab button container.

```
{
  XI_RCT      rct;
  XI_OBJ_DEF* cntrdef;
  XI_OBJ_DEF* btndef;

  rct.top = 3 * XI_FU_MULTIPLE;
  rct.bottom = rct.top + 8 * XI_FU_MULTIPLE;
  rct.left = XI_FU_MULTIPLE;
  rct.right = 43 * XI_FU_MULTIPLE;
  xi_add_rect_def( itfdef, RECT_CID, &rct, XI_ATR_VISIBLE, COLOR_BLACK,
                  COLOR_WHITE );

  rct.bottom = rct.top;
  rct.top = rct.bottom - XI_FU_MULTIPLE;
  cntrdef = xi_add_container_def( itfdef, SECTION_CNTR_CID, &rct,
                                 XI_STACK_HORIZONTAL, FORM_CID );

  btndef = xi_add_button_def( cntrdef, SECTION_ONE_CID, NULL,
                              XI_ATR_ENABLED | XI_ATR_VISIBLE, "Descr",
                              SECTION_TWO_CID );
  btndef->v.btn->type = XIBT_TABBTN;
  btndef->v.btn->checked = TRUE;

  btndef = xi_add_button_def( cntrdef, SECTION_TWO_CID, NULL,
                              XI_ATR_ENABLED | XI_ATR_VISIBLE, "Other",
                              FIELD_BASE_CID + fielddefs[0].type );
  btndef->v.btn->type = XIBT_TABBTN;
}
```

If the **XI_PREF_NATIVE_CTRL**s is set to **FALSE**, XVT radio buttons are used instead of tab buttons.

4.12.9 Default Button

You may want to have a default button in your XI interface. This default button will have a border around it that indicates that it is the default. When the enter key is pressed, an `XIE_BUTTON` event will be generated for the default button. If any button has the focus, it will be the default button. If no button has the focus, you can designate a button as the default. This is done by setting setting `btn_def->v.btn->dflt` to `TRUE` after calling `xi_add_button_def`.

Note that there is an option for the interface that causes enter to work like the tab key. If this is enabled, the enter key will only press a button when a button has focus. The other exception is when a multiline edit field has focus, then pressing enter ends a paragraph.

Only a push button, not radio buttons, check boxes, or tab buttons, may be the default button. In addition, only one push button in the interface may have the `dflt` field set to `TRUE`, for obvious reasons.

Default buttons do not work if you are using native controls. You specify whether you are using native controls or XI controls by setting the preference `XI_PREF_NATIVE_CTRL``SXI_PREF_NATIVE_CTRL`.

The following code, from “`lstdb.c`”, is an example of designating a default button.

```
btndef = xi_add_button_def( cntrdef, CANCEL_BTN_CID, NULL,
                           XI_ATR_ENABLED | XI_ATR_VISIBLE, "Cancel",
                           DEL_BTN_CID );
btndef->v.btn->dflt = TRUE;
```

4.12.10 Drawing in Buttons

You might wish to make a button, or set of buttons, and draw in the buttons. When you draw in the buttons, you can use any of the XI or XVT drawing functions. In this fashion, you can create buttons that are similar to icon buttons, but don't actually use icons.

There is an advantage to using this technique: this technique is completely portable across all XVT platforms. Icon buttons are portable in XI, but the icons themselves are not. However, if you use `xi_draw_line``xi_draw_line`, `xvt_dwin_draw_oval`, etc. to draw in a button, then the button, with your image in it, will be completely portable.

Of course, you could also draw icons when drawing in the button. You could combine drawing icons with other graphics, to create a very sophisticated button. A constraint is that you can only draw into push buttons. Radio buttons, check boxes, and tab buttons do not support this feature.

The technique consists of:

- Setting the `button_def->v.btn->drawable` field in the button definition structure.
- Do drawing during an `XIE_UPDATE` event. This event comes through after XI is done drawing everything that it is going to draw.
- Getting the rectangle from XI for the button that is going to contain the drawing and set the clipping region to that rectangle.
- Calling draw functions.

4.13 Static Text

Static text is simply text used to label objects on the interface such as edit fields. Unlike other XI objects, the user cannot “operate” them, thus the term “static”. They can be right or left justified or made invisible as shown below.

4.13.1 Right-justified Static Text

If a string of static text is visible and has the attribute `XI_ATR_RJUST` set, the text in the string will be displayed as right justified. If the `XI_ATR_RJUST` attribute is not set, the edit field will be left justified.

4.13.2 Enabled/Disabled Static Text

The `XI_ATR_ENABLED` attribute is supported for static text.

The effects of a static text object being disabled is that it is drawn in gray instead of black.

4.13.3 Fonts for Static Text

You may wish for certain static text fields to have a different font than the default for the interface. You change the font used for a static text object by setting `text_def->v.text->font_id` to the desired font after calling `xi_add_text_def`. You will need to create the font using XVT functions. The font is copied by XI during the `xi_create` function and can be destroyed after that function returns.

4.14 Rectangles

You can define rectangles in XI to visually group objects in an interface. When the 3D look is not used, XI draws rectangles with a specified foreground and background color. When the 3D look is used, rectangles are drawn either as wells or platforms, using the colors specified in the preferences

`XI_PREF_COLOR_LIGHT`, `XI_PREF_COLOR_CTRL`, and `XI_PREF_COLOR_DARK`. You specify whether the 3D look is used or not by setting the preference `XI_PREF_3D_LOOK` to `TRUE` or not. The default setting is `FALSE`. The `fore_color` and `back_color` that you specify when calling `xi_add_rect_def` are not used for 3D rectangles. See `xi_set_pref` for details on using preferences.

Rectangle positions are specified in form units. This allows your application to easily place rectangles around XI objects (whose positions are also specified in form units.) It is very important to add your rectangles to your XI interface before adding other types of objects. XI draws the objects in the order added. If you add rectangles after you add other objects such as the form containing edit fields, then the rectangle will be drawn after the edit fields, and obliterate them.

The most common mistake that developers make is to add an XI form, add a rectangle, then add the edit fields to the form. In this case, because of the hierarchy of controls, the form, and thus the edit fields in it get drawn before the rectangle.

You create a rectangle definition by calling `xi_add_rect_def`. The following code, from “`lstlink.c`”, is an example of creating a platform rectangle definition:

```
rct.top = 3 * XI_FU_MULTIPLE;  
rct.bottom = rct.top + 8 * XI_FU_MULTIPLE;  
rct.left = XI_FU_MULTIPLE;  
rct.right = 43 * XI_FU_MULTIPLE;  
xi_add_rect_def( itfdef, RECT_CID, &rct, XI_ATR_VISIBLE, COLOR_BLACK,  
                COLOR_WHITE );
```

4.15 Lines

You may wish to visually divide an XI interface into multiple parts. One way to do this is by creating XI lines. When the 3D look is not used, lines are drawn with a specified foreground color. When the 3D look is used, lines are drawn either as wells or platforms, using the colors specified in the preferences

`XI_PREF_COLOR_LIGHT`, `XI_PREF_COLOR_LIGHT`, `XI_PREF_COLOR_CTRL`, `XI_PREF_COLOR_CTRL`, and `XI_PREF_COLOR_DARK`, `XI_PREF_COLOR_DARK`. You specify whether the 3D look is used or not by setting the preference `XI_PREF_3D_LOOK` to `TRUE` or not. The default setting

is **FALSE**. The **fore_color** and **back_color** that you specify when calling `xi_add_line_def` are not used for 3D lines.

Line positions are specified by two points, in form units. This allows your application to easily place lines between XI objects (whose positions are also specified in form units.)

You create a line definition by calling `xi_add_line_def`.

4.16 Working with XVT/CH

XI has been ported to XVT/CH. However, there is an issue to mention. Some features of XI have to do with manipulation of lists by the mouse. For instance, users may be able to dynamically resize columns by grabbing the column border, and dragging it. Also, users may be able to select columns by clicking on the column heading. All these features also have a programming interface. For example, there is a way to select a column or set the width of the column from your application.

If you need features such as these in your application, you need to give a keyboard interface to the features. Examples of giving a keyboard interface to these features are:

- You may make F4 select the column that contains the cell that has the focus.
- You may make a menu selection to change the width of a column.

4.17 Summary

Now that you know what characteristics XI objects can have, you are ready to learn how the convenience functions are used to set these characteristics and build an object definition tree. With an object definition tree, you can define a complete XI interface. Once you have the definition tree, you can instantiate it by calling `xi_create`. Therefore, creating an XI interface is a two step process: 1) building an object definition tree, and 2) instantiating the interface by passing its object definition tree to `xi_create`.

To give you a preview of where we are heading, in the next chapter, *Defining XI Objects*, you will learn more details about the information contained in object definition structures. Once an interface has been defined you can instantiate it by calling `xi_create` in the context of an XVT application, as mentioned before. This topic is the focus of *Creating an XI Interface*. Once your interface has been defined and created, your application will be notified of user actions and can call XI functions in response. Notification of user actions is the topic of *XI Events*, and manipulating objects is the topic of *Using XI Objects*. Beyond the chapters mentioned are chapters describing application data, tree memory allocation and other advanced topics.

5

Defining XI Objects

5.1 Object Definition Structures

In the beginning of Chapter 3, we introduced the notion that you create an XI object definition tree for the purpose of specifying the available options for an object. This chapter gives you more specific information about these definition structures.

The structures that define an XI object actually consist of two structures attached together. The first structure is of type **XI_OBJ_DEF**. The purpose of this structure is to contain information found in all objects such as a control ID, pointers to children objects and application data. Every object has a control ID which is useful for identifying an object at compile time. For objects who have children, the structure will store the number of children and pointers to an **XI_OBJ_DEF** for each child definition. Application data is an arbitrary piece of data that your application can set for an object.

In addition to containing generic information, a field in the **XI_OBJ_DEF** structure points to a structure that contains information unique to an object. For example, the definition of a button will have an **XI_BUTTON_DEF**, while a list will have an **XI_LIST_DEF**. Together, the general and unique structures contain all the information XI needs to create an object.

The **XI_OBJ_DEF** structure is defined as follows:


```

typedef struct _xi_obj_def
{
    XI_OBJ_TYPE type;
    int cid;
    struct _xi_obj_def *parent;
    short nbr_children;
    struct _xi_obj_def * *children;
    long app_data;
    long app_data2;
    union
    {
        XI_BTN_DEF *btn;
        XI_CONTAINER_DEF *container;
        XI_FORM_DEF *form;
        XI_FIELD_DEF *field;
        XI_GROUP_DEF *group;
        XI_LINE_DEF *line;
        XI_RECT_DEF *rect;
        XI_TEXT_DEF *text;
        XI_COLUMN_DEF *column;
        XI_ITF_DEF *itf;
        XI_LIST_DEF *list;
    } v;
} XI_OBJ_DEF;

```

Refer to the *XI Programmer's Reference* for detailed information about the contained structures (**XI_ITF_DEF**, **XI_FORM_DEF**, **XI_FIELD_DEF**, etc.)

Because the unique structures differ for each type of object, a different convenience function is used to define each type of object. In the rest of this chapter, we will outline the relationship between an object's definition structures and the arguments to its convenience function. Keep in mind that when writing a real application with XI, you would seldom define an individual object. Instead, you would normally define a tree describing all of the objects in an interface. However, by looking at each object as it would be defined individually, we can focus on the relationships between fields of the definition structures and arguments to the convenience functions without "worrying about the relatives."

5.2 Defining an Interface Object

As mentioned above, the **XI_OBJ_DEF** portion of any object definition has a control ID, application data, number of children, a pointer to an array of pointers to those children and a pointer to a unique structure, an **XI_ITF_DEF** in this case.

In the **XI_ITF_DEF** portion of the interface definition, you will specify an event handler of type **XI_EVENT_HANDLER**. This is where you connect an event handler to the XI interface.

Also in the **XI_ITF_DEF** structure, you will find a rectangle which is the bounding rectangle for the interface and the title of the XVT window, as well as booleans for the presence of a size box, vertical scroll bar, horizontal scroll bar and close box.

When you use the convenience function, **xi_create_itf_def**, to create the definition structures for an interface, you will need to pass in a control ID, the event handler, a pointer to a rectangle, a pointer to the title string and application data stored in a **long**.

The bounding rectangle can be set in one of three ways. You can explicitly set the rectangle so that the XVT window created by XI will be the size of the rectangle. Alternatively, you can tell XI to automatically size and position the window by setting the dimensions of the rectangle to zero. By specifying the dimensions of the upper left-hand corner of the rectangle, and then setting the bottom right coordinate to the same value (an empty rectangle), you can determine the location of the window, but have XI automatically size it for you.

When looking at the list of arguments to `xi_create_itf_def`, you might have also noticed that options for the window features such as the size box, scroll bars and close box aren't included in the list. By default, the window will have a close box but not have scroll bars and not be resizable. If you want different options, you will need to set these in the `XI_ITF_DEF` structure after calling `xi_create_itf_def`. To do this, you get a pointer to the `XI_OBJ_DEF` returned by `xi_create_itf_def`. Recall that the `XI_OBJ_DEF` contains a pointer to the `XI_ITF_DEF` where you can set these booleans as shown in the following code from "lstdb.c".

```
itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)form_eh, NULL,
                          "Delete Employee", 0L );
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->modal = TRUE;
```

You might have also noticed that `xi_create_itf_def` did not take either the number of children or a pointer to an array of them. As we saw in section 3.2, *Using the Convenience Functions*, the convenience functions are responsible for adding children to the array of objects in the children's parent.

5.3 Defining Forms

Like other XI objects, the `XI_OBJ_DEF` portion of any object definition has a control ID, application data, number of children, a pointer to an array of pointers to those children and a pointer to a unique structure, an `XI_FORM_DEF` in this case.

In `XI_FORM_DEF`, you will find a tab control ID which points to the control ID of the next composite object in the interface such as a list, container, or another form. The tab control ID is used to control how the focus moves when the user presses the "meta-tab" key. When pressed, this key tells XI to move the focus to the next composite object in the navigation sequence—allowing the user to move between objects easily.

When you looked at the list of items a form definition contains, you might have noticed that the form has no placement or sizing information. This is because it is only a collection of edit fields. The edit fields in a form can be placed anywhere inside the window.

When using the convenience function, `xi_add_form_def`, to define a form you will pass in a control ID, a tab control ID and a pointer to the interface definition. The form becomes the child of the interface definition.

When looking at the list of arguments to `xi_add_form_def`, you might have noticed that you didn't pass in a **long** for application data. Although you can supply application data for all XI objects, you will need to explicitly set the application data field in the `XI_OBJ_DEF` structure for any object other than the interface.

The following code, from "lstdb.c", shows the definition of a form.

```
formdef = xi_add_form_def(itfdef, FORM_CID, FORM_CID );
```

5.4 Defining Edit Fields

In the `XI_OBJ_DEF` for an edit field, you will find the control ID for the edit field, a place to keep application data, and a pointer to the unique structure for an edit field, `XI_FIELD_DEF`. In the `XI_OBJ_DEF`, there are places to hold information about children, but since edit fields do not have children, these fields will be set to `NULL`.

In the `XI_FIELD_DEF` structure, you can set the tab control ID, edit field position and width, number of characters the user can type, the attribute, and enabled and disabled colors, background and disabled background colors, and the color of the edit field when it has the focus. The `XI_FIELD_DEF` structure contains many other important pieces of information. See `XI_FIELD_DEF` in the XI Programmer's Reference for a complete list of the fields available in an XI edit field.

The control ID for an edit field is important in specifying where the edit field is in the tabbing sequence. The tab control ID determines the next edit field or other object that can actually have focus. You should not set this to tab to a form, container or list.

The convenience function you use to define an edit field is `xi_add_field_def`. When calling `xi_add_field_def`, you pass in much of the information mentioned above. Many features need to be enabled by setting values explicitly after the edit field is defined.

To set the position of the edit field, you specify its upper left-hand corner in form units. You set the text size and the edit field width separately because an edit field can contain more characters than can be displayed if it has the `XI_ATR_AUTOSCROLL` attribute set.

The following code, from “`lstlink.c`”, shows creating edit field definitions and setting values in the definition structure.

```
for ( num = 0; fielddefs[ num ].width != 0; num++ )
{
    XI_OBJ_DEF* def;
    long attrib;

    attrib = XI_ATR_AUTOSELECT | XI_ATR_AUTOSCROLL;
    if ( fielddefs[ num ].section <= 1 )
        attrib |= XI_ATR_VISIBLE;
    if ( fielddefs[ num ].enabled )
        attrib |= XI_ATR_ENABLED | XI_ATR_BORDER;
    def = xi_add_field_def( formdef, FIELD_BASE_CID
                          + fielddefs[ num ].type,
                          fielddefs[ num ].v * XI_FU_MULTIPLE,
                          fielddefs[ num ].h * XI_FU_MULTIPLE,
                          fielddefs[ num ].width * 3
                          / 2 * XI_FU_MULTIPLE,
                          attrib, ( fielddefs[ num + 1 ].type == 0 )
                          ? SAVE_BTN_CID : FIELD_BASE_CID
                          + fielddefs[ num + 1 ].type,
                          fielddefs[ num ].width + 1, COLOR_BLACK,
                          COLOR_WHITE, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK );
    if ( fielddefs[ num ].height != 0 )
    {
        XI_RCT* prct = &def->v.field->xi_rct;

        prct->top = def->v.field->pnt.v;
        prct->left = def->v.field->pnt.h;
        prct->bottom = prct->top + fielddefs[ num ].height *
        XI_FU_MULTIPLE;
        prct->right = prct->left + 40 * XI_FU_MULTIPLE;
    }
    def->v.field->button = fielddefs[ num ].button;
}
}
```

5.5 Defining Lists

As with other XI objects, the general structure for a list definition has a control ID, a place to store application data, the number of columns in the list, and a pointer to an array of pointers to those columns.

As with other XI objects, the tab control ID indicates the next object in the tabbing sequence. In this case, it is the next composite object (list, form or container) to receive the focus when the user presses the meta-tab key. The tab control ID is found in the structure `XI_LIST_DEF`.

Also in `XI_LIST_DEF`, you will find information about the location of the list, its height, its attributes and colors. The location and height are in form units. The colors are the enabled color, background color, disabled color, disabled background color and active color. The active color determines the color to which a cell changes when it gains the focus. The `XI_LIST_DEF` structure contains many other important pieces of

information. See **XI_LIST_DEF** in the XI Programmer's Reference for a complete list of the fields available in an XI list.

When looking at the items in the list definition structures, it is not necessary to set a width for the list. This is because XI will calculate the width of the list automatically by adding together the widths of the columns with an appropriate amount of spacing between them. Because the width of a column is stored in the column's unique definition structure, (**XI_COLUMN_DEF**) and the general definition structure for the list (**XI_OBJ_DEF**) points to an array of pointers to the definitions of its columns, this information is available to XI so that it can determine the width of the list. However, if you do set a width for the list, then XI will scroll columns horizontally within that width.

You might have also noticed that you didn't set the number of rows for a list. This is because XI will calculate the number of rows that will fit inside the list height at run-time when the list is instantiated. You might also have noticed that you can specify a one-row list explicitly, and XI will automatically size it for you. This is because a list that is one row on one platform may be three rows on another. If having the list be one row is important to the look and feel of your application, you can force it by overriding XI's automatic row fitting behavior.

The convenience function used to define a list is `xi_add_list_def`. Its arguments correspond to the fields in the definition structures. You can set many other options for a list after the convenience function has created the appropriate definition structures. The following code, from "lstdb.c", demonstrates creating a list definition and setting options for it.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0, 8 * XI_FU_MULTIPLE,
                          XI_ATR_ENABLED | XI_ATR_VISIBLE |
                          XI_ATR_TABWRAP, COLOR_BLACK, COLOR_WHITE,
                          COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                          LIST_CID );
listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

5.6 Defining Columns

Like other XI objects, the general information for a column definition is stored in its **XI_OBJ_DEF** structure. This information is the column's control ID, application data, and information about children. Since columns do not have children, these fields are set to **NULL** by the convenience function that defines a column, `xi_add_column_def`.

In the unique definition structure, **XI_COLUMN_DEF**, you will find a position, column width, buffer or text size, heading text and column attributes. The **position** field tells XI the order of the columns in the list. It is important because you can add (or delete) columns after the list has been instantiated. This number will determine where the new column will go.

The column width is the size of the column in form units. Notice that it doesn't correspond to the number of characters the user can type in a cell. This is because string length is determined by the buffer size. If the attribute, **XI_ATR_AUTOSCROLL** is set for the column, the user can type more characters than the cell will display at one time.

The heading text is the label at the top of the column.

The arguments for the convenience function, `xi_add_column_def` that match the fields of the structures mentioned above are the control ID, attribute, sort number, width, text size and column heading text. Once again, application data and other options are not included and need to be set explicitly after the function has returned.

The following code, from "lstmem.c", demonstrates creating a column definition and setting options for it.

```
coldef = xi_add_column_def( listdef, COL_BASE_CID + VALUE_ITEM_NBR,
                           STD_COL_ATR | XI_ATR_SELECTABLE,
                           1, 6 * XI_FU_MULTIPLE, 5, "Nbr" );
#if THREE_DIMENSIONAL
coldef->v.column->heading_platform = TRUE;
coldef->v.column->column_platform = TRUE;
#endif
coldef->v.column->size_rows = TRUE;
```

5.7 Defining Containers

Like other XI objects, the **XI_OBJ_DEF** portion of any object definition has a control ID, application data, number of children, a pointer to an array of pointers to those children (button definitions) and a pointer to an unique structure, an **XI_CONTAINER_DEF** in this case.

The bounding rectangle is set in form units. The container orientation specifies whether the buttons are stacked horizontally, vertically, or in a grid. The tab control ID determined the next composite object (form, list or container) in the tabbing sequence for the meta-tab key. All of these are available as arguments to the convenience function, `xi_add_container_def`. As with other objects, application data for the container will need to be set explicitly after `xi_add_container_def` returns.

The members of the **XI_CONTAINER_DEF** structure include a packed flag and height and width for the size of buttons in a "grid" orientation. The following code, from "lstdb.c", shows how to create a container definition and set options for it.

```
rct.top = 0;
rct.left = 0;
rct.bottom = 24;
rct.right = 144;
cndef = xi_add_container_def( itfdef, CONTAINER_CID, &rct,
                             XI_GRID_HORIZONTAL, LIST_CID );
cndef->v.container->packed = TRUE;
cndef->v.container->btn_width = 6 * XI_FU_MULTIPLE;
```

5.8 Defining Buttons

In the general structure defining a button, you will find the button's control ID, application data, and information about its children. Since a button does not have children, these fields are set to **NULL** by the convenience function, `xi_add_button_def`.

In the structure unique to button definitions, **XI_BTN_DEF**, you will find its bounding rectangle, attribute, label text and tab control ID.

The bounding rectangle is in form units and specifies how big you want the button to be up to the size of the container holding it. The tab control ID specifies the next button in the tabbing sequence. Unlike the tab control IDs for composite objects, this determines what button the focus will move to when the user presses the **tab** key. Some of the information used to define a button is available as arguments to the convenience function `xi_add_button_def`. As with other objects, other features will need to be enabled by setting fields explicitly when `xi_add_button_def` returns

The following code, from "lstdb.c", shows how to create a button definition and set options for it.

```
btndef = xi_add_button_def( cndef, DEL_BTN_CID, NULL,
                           XI_ATR_ENABLED | XI_ATR_VISIBLE, "Del",
                           ADD_BTN_CID );
btndef->v.btn->fore_color = COLOR_RED;
btndef->v.btn->down_icon_rid = DEL_BTN_ICON;
btndef->v.btn->up_icon_rid = DEL_BTN_ICON;
```

5.9 Defining Groups

As you've seen before, in the **XI_OBJ_DEF** structure to define an XI object, you will find the control ID of the object, a place to put application data, number of children and a pointer to an array of pointers to the definitions of those children. With groups, you don't actually have children because the group merely refers to the children of another object (list or form). This is because you wouldn't want the same object instantiated twice. Therefore, the references made to the members of the group are found in the structure unique to groups, **XI_GROUP_DEF**.

The references made to the members of a group are to the control IDs of the edit fields or columns in the group. Keep in mind that you cannot have a group with both edit fields and columns. Also in the **XI_GROUP_DEF** structure is the number of members. There are no attributes for a group.

All characteristics of a group are set via the convenience function, **xi_add_group_def** except for application data as with other interface children.

The following code, from "lstdb.c", shows how to create a group definition.

```
{
    int cids[2];

    cids[0] = COL_BASE_CID + DB_EMP_MINHRS;
    cids[1] = COL_BASE_CID + DB_EMP_MAXHRS;
    xi_add_group_def( itfdef, GROUP_CID, 2, cids );
}
```

5.10 Defining Static Text

As with other objects that do not have children, the children fields in the **XI_OBJ_DEF** structure to define all objects are not used when defining static text objects.

In the unique structure, **XI_TEXT_DEF**, the bounding rectangle for the text object can be set as well as its label. In addition, you can set its attribute. Setting these fields is done with the the convenience function, **xi_add_text_def**. Like other objects that are children of an interface, application data for the text object must be set explicitly.

The following code, from "lstdb.c", shows how to create text definitions.

```
for ( num = 0; textdefs[ num ].text != NULL; num++ )
{
    XI_RCT rct;

    rct.top = textdefs[ num ].v * XI_FU_MULTIPLE;
    rct.left = textdefs[ num ].h * XI_FU_MULTIPLE;
    rct.bottom = rct.top + XI_FU_MULTIPLE;
    rct.right = rct.left + strlen( textdefs[ num ].text )
                * 3 / 2 * XI_FU_MULTIPLE;
    xi_add_text_def( itfdef, TEXT_BASE_CID + num, &rct,
                    XI_ATR_VISIBLE | XI_ATR_ENABLED,
                    textdefs[ num ].text );
}
```

5.11 Defining Rectangles

As with other objects that do not have children, the children fields in the **XI_OBJ_DEF** structure to define all objects are not used when defining rectangle objects.

In the unique structure, **XI_RECT_DEF**, the bounding rectangle for the rectangle can be set, as well as colors, and whether the rectangle is a well or platform rectangle. In addition, you can set its attribute. Setting most of these fields is done with the the convenience function, **xi_add_rect_def**. The 3D appearance of rectangles (i.e. well or platform) is enabled by setting the **well** field after the return of **xi_add_rect_def**.

The following code, from “lstlink.c”, shows how to create a rectangle definition.

```
rct.top = 3 * XI_FU_MULTIPLE;  
rct.bottom = rct.top + 8 * XI_FU_MULTIPLE;  
rct.left = XI_FU_MULTIPLE;  
rct.right = 43 * XI_FU_MULTIPLE;  
xi_add_rect_def( itfdef, RECT_CID, &rct, XI_ATR_VISIBLE, COLOR_BLACK,  
                COLOR_WHITE );
```

5.12 Defining Lines

As with other objects that do not have children, the children fields in the **XI_OBJ_DEF** structure to define all objects are not used when defining line objects.

In the unique structure, **XI_LINE_DEF**, the starting and ending points of the line can be set, as well as colors, and whether the line is a well or platform line. In addition, you can set its attribute. Setting these fields is done with the convenience function, **xi_add_line_def**.

5.13 Summary

In the next chapter, you will learn how to instantiate a definition tree by calling **xi_create**. Once the hierarchy is instantiated, you may not need the definition tree any longer. You can free it by calling the **xi_tree_free** or **xi_def_free** function with the pointer to the definition tree. However, if your application allows closing and reopening of interfaces or allows two instances of the same interface to be active at the same time, you may want to keep the definition tree around so that you don't have to redefine it each time you need to instantiate the interface.

6

Creating an XI Interface

creating an interfaceBefore you can instantiate an XI interface, you will need to create an interface definition tree as explained in the chapter *Creating an Object Definition Tree*.

After you construct an object definition tree, you will need to call `xi_create` to instantiate it. In addition to creating the interface, you will need to size it and hook it up to XVT. These topics are the focus of this chapter.

6.1 Sizing the Interface

As mentioned before, in order to create an interface, your application must construct a definition for the interface, and then instantiate it by calling `xi_create`. However, there can be an intermediate step between defining the tree and instantiating it whereby your application is given the chance to adjust the size and position of the window that will be created to hold the interface. Since each native system has a different size of screen, height of the system font, border widths and other display characteristics, your interface will be a different size on each system. Because you cannot know how big an interface is going to be at compile time, it is important to have XI tell you how big the interface will be on the system before you instantiate it.

After creating your interface definition tree, your application can call the function `xi_get_def_rect` to get the bounding rectangle that the interface will occupy. This rectangle is in pixels, not form units, and its upper-left corner is always zero. Given this rectangle, it is possible for your application to compute an appropriate rectangle for the window to hold the interface.

There are three approaches you can take to size the interface. The first approach is to let XI figure out how big the window should be to hold the interface. This is done by specifying a **NULL** rectangle in the `rectp` field of `XI_ITF_DEF` structure in the object definition tree. When using the convenience functions, pass **NULL** for the `rectp` argument. This tells XI to compute the window size and position. It does this by figuring the bounding rectangle around all controls in the interface and then adding a margin along the bottom and right, according to the preferences `XI_PREF_ITF_WS_BOTTOM` and `XI_PREF_ITF_WS_RIGHT` or the local overrides in the interface definition, `whitespace_bottom` and `whitespace_right`.

The second approach is to let XI decide how large the window should be, but you decide where the upper left corner of the window should be placed. To do this, in the **rctp** field of the **XI_ITF_DEF** structure, you will specify an empty rectangle by setting the top and left fields, and then setting the right field equal to the left and the bottom equal to the top. This tells XI where to position the window on the screen, but it will need to compute the size itself.

The third approach is to determine the size and placement of the window yourself. To do this, you will probably call **xi_get_def_rect** to get the dimensions of the rectangle in form units. You then compute your own rectangle based on that, and put the rectangle you compute into the **rctp** field of the **XI_ITF_DEF** structure. The third approach is the most flexible because among other things, it allows you to reserve some extra space in the window for your application's own use.

The following code, from "main.c", shows how to set the interface definition rectangle so that the interface will be centered.

```
void center_interface( XI_OBJ_DEF* itfdef )
{
    RCT r;
    int width, height;
    RCT* itf_rct = itfdef->v.itf->rctp;

    xi_get_def_rect( itfdef, &r );
    width = r.right - r.left;
    height = r.bottom - r.top;
    xvt_vobj_get_client_rect( TASK_WIN, &r );
    itf_rct->top = ( r.bottom - height ) / 2;
    if ( itf_rct->top < xi_get_pref( XI_PREF_ITF_MIN_TOP ) )
        itf_rct->top = (int)xi_get_pref( XI_PREF_ITF_MIN_TOP );
    itf_rct->left = ( r.right - width ) / 2;
    if ( itf_rct->left < xi_get_pref( XI_PREF_ITF_MIN_LEFT ) )
        itf_rct->left = (int)xi_get_pref( XI_PREF_ITF_MIN_LEFT );
    itf_rct->bottom = itf_rct->top;
    itf_rct->right = itf_rct->left;
    if ( itfdef->v.itf->modal )
        xvt_vobj_translate_points( TASK_WIN, SCREEN_WIN, (PNT*)itf_rct,
2 );
}
```

6.2 Instantiating the Interface

After you have determined the size of the interface, or plan to let XI determine it for you, you are ready to call **xi_create** with the **XI_OBJ_DEF** for the interface definition. In addition to the pointer to the interface definition tree, **xi_create** takes another argument which is the parent. When instantiating an entire interface, you set the parent parameter to **NULL** to indicate that you are going to create a complete interface hierarchy. **xi_create** will return an **XI_OBJ** pointer to the interface.

Once the hierarchy is instantiated, you may not need the definition tree any longer. You can free it by calling the **xi_tree_free** or **xi_def_free** function with the pointer to the definition tree. However, if your application allows closing and reopening of interfaces or allows two instances of the same interface to be active at the same time, then you may want to keep the definition tree around so that you don't have to redefine it each time you need to instantiate the interface.

At this point, you know how to define an interface and instantiate it. Having the interface work correctly in the context of an XVT application involves XVT programming as explained in the next section.

6.3 Hooking It Up to XVT

In addition to the code you need to write with XI, there are two XVT functions you must write in order to create an XI application. These are **main** and the task window's event handler. These functions are necessary when writing any XVT application, including one using XI. In this section, you will learn how to

program these functions in the context of an XI application. If, after reading this section, you need more information about **main** and the task window's event handler, see the XVT documentation.

6.3.1 Programming the Task Window's Event Handler

There are two things you must do within the task window to have XI work properly. You might want to call **xi_create** in response to a user action such as a menu item being selected. In addition to responding to native XVT events, the second thing you must do in the task window's event handler is to put in a call to **xi_initxi_init** on the **E_CREATE** event, so that XI can initialize itself. (There is no cleanup operation on XI.)

The first half of “main.c” demonstrates how to do this. The function **main** makes the proper call to initialize XVT. It sets the task window handler function to **task_eh**. This function calls **init_application** which handles the initialization of XI. The **task_eh** function also calls **do_menu** which calls the various functions that create XI interfaces.

6.3.2 Connecting XI Interfaces to XVT

There are three different methods by which you hook up an XI interface to XVT. The following three sections examine these different methods. The last section explains a variation on the third method, to place an XI interface into the task window.

The fundamental issue with hooking up XI to XVT is to make sure that the appropriate XVT events are getting to XI. There is nothing hidden about how XI works. XI simply processes XVT events, but it is important that the XVT events get to XI. It is helpful for you to know exactly how events are getting to XI.

If you have problems getting a window to have the behavior that you want, do the following experiments:

- Make sure that you can create the window in XVT, with the characteristics that you want. Initially, don't put the XI interface into the window - leave XI out of the picture. After you can create the XVT window exactly like you want it, then it is straightforward to place the XI interface into the existing window.
- Verify that events are getting to XI. If you have your own XVT event handler for the window, place a break point immediately before the call to **xi_event**.
- One other common problem is when **xi_init** does not get called by the application before the XI system font is set or preferences are set. Verify that **xi_init** was called. XI will call **xi_init** for you, if you call **xi_create** before calling **xi_init**, but it is better not to rely on this because the XI preferences and system font may not be set the way you want.

6.3.3 Approach 1: Let XI Create the Window For You

By default, when you call **xi_create**, and let XI create the window for the XI interface, then XI uses **xi_event** as the XVT event handler for the window. This means that when XVT generates an event, it calls **xi_event** to process it. XI then does three main things with the XVT event:

1. XI calls your XI event handler with an **XIE_XVT_EVENT**. This gives you the opportunity to process XVT events and, perhaps, refuse them so that XI does not process the event.
2. If it is not refused, XI then processes the XVT event and may generate one or more XI events. For example, an **E_MOUSE_DOWN** event may cause several XI off and on focus events.
3. Finally, XI generates an **XIE_XVT_POST_EVENT** for the XVT event. This gives you an opportunity to do any processing that might be necessary after XI has completed processing the XVT event.

Most of the interfaces in the example use this method. The following code, from “lstdb.c”, demonstrates this method of creating an interface.

```

itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh, NULL,
                          "Linked List", 0L );
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_BAR_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;

... /* Other definitions created here. */

xi_create( NULL, itfdef );
xi_dequeue();
xi_tree_free( itfdef );

```

6.3.4 Approach 2: Create the XVT Window with `xi_event` as Event Handler

This approach uses the XVT function to create the window and then sets the `win` field of the interface definition. This tells XI to use the existing window instead of creating its own. The `xi_event` function is still used as the direct XVT event handler, but must be explicitly specified in the XVT function that creates the window.

The following code, from “`lstdb.c`”, shows how to do this in order to make the window iconizable.

```

{
    RCT r;

    xi_get_def_rect( itfdef, &r );
    xvt_rect_offset( &r, (short)xi_get_pref( XI_PREF_ITF_MIN_LEFT ),
                   (short)xi_get_pref( XI_PREF_ITF_MIN_TOP ) );
    itfdef->v.itf->win = xvt_win_create( W_DOC, &r, "Employee List",
                                       MENU_BAR_RID, TASK_WIN,
                                       WSF_SIZE | WSF_CLOSE
                                       | WSF_ICONIZABLE, EM_ALL,
                                       (EVENT_HANDLER)xi_event, 0L );
}
xi_create( NULL, itfdef );
xi_dequeue();
xi_tree_free( itfdef );

```

While the prototype for `xi_event` is nearly identical to the prototype of `EVENT_HANDLEREVENT_HANDLER`, it is not the same. Therefore, cast `xi_event` as an `EVENT_HANDLER` when calling `xvt_win_create`.

6.3.5 Approach 3: Create the XVT Window with Your Own Event Handler

The third approach is to write an event handler for your XVT window. In this event handler, you must call `xi_event`, passing all events to XI for processing. This approach is very rare since all XVT events can be processed in the XI event handler. It’s your choice as to when to call `xi_event`, but any XVT event that does not get passed to XI in this way may cause the XI interface to function incorrectly.

6.3.6 Putting an XI Interface in the Task Window

In XVT/Win, XVT/NT or XVT/PM, you may wish to put an XI interface into the task window. In order to do this, four things must be done:

1. The task window must be made drawable. This is done by calling `xvt_vobj_set_attr` with the appropriate attribute. This attribute can be found in the platform-specific books from XVT. The attribute will probably be `ATTR_WIN_PM_DRAWABLE_TWIN`. This function must be called before initializing XVT.
2. You must place a call to `xi_event` in the task window event handler. This is usually added at the end of the XVT event handler function.
3. After you have initialized XI in the `E_CREATE` event for the task window, you can then create an XI interface and set the `win` field of the interface definition to `TASK_WIN`.

Some notes about drawable task windows:

- If you do not create the XI interface during the `E_CREATE` event, you are responsible for handling the background drawing of the task window.
- The MDI (multiple document interface) cannot be used with a drawable task window.

In general, we suggest that you avoid drawing into the task window because it is not portable to all platforms.

6.4 Summary

After you have done the necessary things to create an interface, and program `main` and the task window's event handler, the next thing you must do is write an XI event handler function. The purpose of the event handler function is to handle the XI events that are generated as the user operates the interface. XI events and the event handler functions you'll need to write are described in the following chapter.

7

XI Events

In this chapter, you will encounter an overview of XI events and a description of the code you'll need to write to handle them. When programming with XI, you will need to create an XI event handler function to receive the events generated for each interface. Recall that XI will generate events to notify your application that the user has done something, or to ask it for information. Following the discussion on event handlers, we will describe in more detail the types of events XI generates, and how the events are sent to your event handler. In addition, you will find a discussion of the XI focus model and the events associated with focus notification.

7.1 XI Event Handlers

An XI event handler is the focal point for all of the activity that takes place in an XI application during the lifetime of an interface. XI generates the appropriate events by passing an **XI_EVENT** structure to your XI event handler function. The event structure contains a field indicating the type of event. In addition, the event structure contains any pertinent data related to the event. After receiving the event, the event handler is responsible for figuring out whether or not the event is important, and then acting on it appropriately. For example, the event handler may be asked to provide XI with data for a list; it may want to take some action to respond; or it may want to refuse the event as the case may be. As you can see, the event handler you will be writing will need to be prepared to receive any event XI might send throughout the lifetime of the interface.

To illustrate, as soon as an interface is instantiated, your event handler will be informed so that it can set up its data structures and allocate memory, open files, or whatever else it needs to do at that time. If the interface has a form or list, your application will also need to initialize it soon after the interface is instantiated.

Once the interface is fully initialized, your event handler will begin receiving events notifying it that the user is manipulating controls, and informing it that the user is finished with what he is doing and wants to do something else on the interface. When appropriate, your event handler can refuse these events to prevent these actions from taking place. When the user is done using the interface, the event handler will receive an event so that it can free up memory, destroy XVT fonts and so on.

To set up an event handler, you must designate a function to receive XI events, and register this function with XI when defining the interface. In addition, by passing XVT events to XI via `xi_event`, you

are telling XI to intercept XVT events, turn them into XI events, and send the XI events to your event handler.

An application can use the same XI event handler for all of its interfaces or it can have a different one for each interface. It doesn't really matter as long as the application supplies some event handler for each XI interface that your application creates.

7.2 Responding to XI Events

XI events are sent to your application via an event handler, as explained in the introduction to this chapter. An event handler has two parameters. The first parameter is a pointer to the interface object. The second parameter is an `XI_EVENT` structure. The `XI_EVENT` structure will contain information about the event such as the type of event and data your application might need in order to respond to the event.

When writing your event handler function, you will need to create a mechanism for separating events by event type. To separate the events by type, you will need to switch on the `type` field of the event structure passed to your event handler so that for each type of event your event handler receives, your program does something different. The types of events are things like `XIE_INIT`, `XIE_OFF_FIELD`, `XIE_ON_FORM`, `XIE_CHG_CELL`.

Within the code that responds to each event type, you may need to further subdivide the event handling code so that different code is executed for each object you care about. Once again, this is typically done with a switch statement. Within an event structure, many events have an `XI_OBJ` associated with them where you can find out the control ID of the object for which the event was generated. You switch on the control IDs of the objects you might have in your interface so that you can have a separate case in your event handler function for each object. For example, on an `XIE_BUTTON` event, you could have a switch statement where there is one case for every button in the interface.

As you'll see in the following code example, we are illustrating only the mechanics of responding to XI events. What the events mean and why they are sent are topics addressed later in this chapter. What your application should do in response to receiving an event is discussed in detail in the next chapter, *Using XI Objects*. The following code example illustrates using switch statements to control program flow in an XI event handler.

```
void form_process_button( XI_OBJ* itf, XI_OBJ* button )
{
    FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

    switch ( button->cid )
    {
        case SAVE_BTN_CID:
            if ( !xi_move_focus( itf ) )
                break;
            if ( form_info->changed )
                update_record( xi_get_obj( itf, FORM_CID ), form_info-
>handle );
            refresh_row( form_info->parent_list, form_info->handle );
            form_info->changed = FALSE;
            xi_delete( itf );
            break;
        case CANCEL_BTN_CID:
            if ( form_info->changed && xvt_ask( "No", "Yes", NULL,
                "You have made changes. Are you sure you want to cancel?" )
                != RESP_2 )
                break;
            xi_delete( itf );
            break;
    }
}
```

```

case SECTION_ONE_CID:
case SECTION_TWO_CID:
{
    SECTION_INFO* info = (SECTION_INFO*)xi_get_app_data( button );

    if ( form_info->cur_section != info )
    {
        change_section( form_info->cur_section, FALSE );
        change_section( info, TRUE );
        form_info->cur_section = info;
    }
    xi_check( button, TRUE );
    break;
}
case FIELD_BASE_CID + LINK_WHO:
    create_who_list( button, form_eh );
    break;
}
}

static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
    FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

    switch ( xiev->type )
    {
        case XIE_INIT:
        {
            int      num;
            XI_OBJ*  form;
            XI_OBJ** field;

            form = xi_get_obj( itf, FORM_CID );
            field = form->children;
            for ( num = 0; num < form->nbr_children; num++, field++ )
                link_set_text( form_info->handle, (*field)->cid
                    - FIELD_BASE_CID, *field );

            break;
        }
        case XIE_CLOSE:
            if ( ( !xi_move_focus( itf ) || form_info->changed )
                && xvt_ask( "No", "Yes", NULL,
                    "You have made changes. Are you sure you want to cancel?" )
                != RESP_2 )
                xiev->refused = TRUE;
            break;
        case XIE_BUTTON:
            form_process_button( itf, xiev->v.xi_obj );
            break;
        case XIE_CHG_FIELD:
            form_info->field_changed = TRUE;
            break;
    }
}

```

```

case XIE_OFF_FIELD:
    if ( form_info->field_changed )
    {
        XI_OBJ* field = xiev->v.xi_obj;

        if ( !link_validate( field->cid - FIELD_BASE_CID,
            xi_get_text( field, NULL, 0 ) ) )
            xiev->refused = TRUE;
        else
        {
            form_info->changed = TRUE;
            form_info->field_changed = FALSE;
        }
    }
    break;
}
}
}

```

7.3 Refusing XI Events

refusing events Unlike other windowing systems, XI often notifies you of an event before it happens so that you can prevent the action from taking place. For example, XI has a class of events called focus events which are sent to inform your application that the user is trying to move keyboard focus from one control to another. Typically, a focus event is a signal to your application indicating that it should validate the contents of an edit field, form, cell, or row before letting the user continue. If the data entered is invalid, you can set the **refused** flag in the XI event structure to **TRUE** and return to XI. This will keep XI from moving the focus until the user has typed valid data.

Not all XI events can be refused. In particular, you cannot refuse events sent to indicate that something has already happened. An example of a non-refusable event is **XIE_CLEANUP** which informs your application that an interface has been deleted so that it can free up the memory used for application data associated with the interface. This event cannot be refused because it is an indication that the interface has already been deleted. However, the **XIE_CLOSE** event, which occurs just before the interface is deleted, can be refused. Most events that cannot be refused have a corresponding event that can be refused. These relationships are detailed in the *XI Programmer's Reference*.

The following tables list events that can be refused and those that can't. Each event is described briefly in the tables and then described in more detail later in the chapter.

Refusable Events

XIE_XVT_EVENT	XI sends every XVT event to your XI event handler before XI processes that event. You refuse the event if you want XI to ignore that particular XVT event.
XIE_COL_DELETE	XI sends this interface event to inform your application that the user attempted to delete a column by dragging it off of the list. You refuse the event if you do not want the column to be deleted.
XIE_COL_MOVE	XI sends this interface event to inform your application that the user moved a column. You refuse the event if you want the column to remain in the same position.
XIE_COL_SIZE	XI sends this interface event to inform your application that the user sized a column. You refuse the event if you want the column to remain the same width.
XIE_CLOSE	XI sends this interface event to inform your application that the user attempted to close the window containing the interface. You refuse the event if you want the user to finish doing something else before deleting the interface.

XIE_DROP_ROW	XI sends your application this row event to notify it that the user attempted to move a row by dragging and dropping it. You refuse the event if you do not want the row moved.
XIE_GET_FIRST	This record event is sent by XI to ask that your application get the first record displayed in a list. This event is most often seen when initializing a list. You refuse the event if there is no record.
XIE_GET_LAST	This record event is sent by XI to ask that your application get the last record displayed in a list. This event is most often seen when you move the thumb to the bottom of the vertical scroll bar. You refuse the event if there is no record.
XIE_GET_NEXT	This record event is sent by XI to ask your application to get the next record displayed in a list. This event is most often seen when initializing a list and during a downward scrolling operation. You refuse the event if there is no record.
XIE_GET_PREV	XI asks your application to get the previous record displayed in a list. This event is most often seen during an upward scrolling operation. You refuse the event if there is no record.
XIE_OFF_CELL	XI sends XIE_OFF_* focus events to inform your application that the user is attempting to move the focus off of an XI object. Reasons for refusing focus events are described later in this chapter and in <i>Using XI Objects</i> .
XIE_OFF_COLUMN	See XIE_OFF_CELL .
XIE_OFF_FIELD	
XIE_OFF_FORM	
XIE_OFF_GROUP	
XIE_OFF_LIST	
XIE_OFF_ROW	
XIE_CHAR_CELL	This event is sent by XI to notify your application that the user has typed in a cell. You refuse this event if you don't want the character to be inserted into the cell.
XIE_CHAR_FIELD	This event is sent by XI to notify your application that the user has typed in an edit field. You refuse this event if you don't want the character to be inserted into the edit field.
XIE_ON_CELL	XI sends XIE_ON_* events to inform your application that the user is attempting to move the focus to an object on the interface. You refuse the event if you want the user to finish doing something else on the interface before moving to the object.
XIE_ON_COLUMN	See XIE_ON_CELL .
XIE_ON_FIELD	
XIE_ON_FORM	
XIE_ON_GROUP	
XIE_ON_LIST	
XIE_ON_ROW	
XIE_ROW_SIZE	XI sends this event when the user dynamically changes the height of a row. You refuse the event if you want the row height to remain as it was.
XIE_SELECT	XI sends this event to inform your application that the user selected or deselected either a row, a column, or a range of cells on the list. You

refuse the event if you want the row or column not to be selected or deselected. When selecting (or deselecting) a range of cells, the **XIE_SELECT** event is not refusable.

non-refusable events**Non-Refusable Events**

XIE_XVT_POST_EVENT	XI sends every XVT event to your XI event handler after XI processes the event.
XIE_BUTTON	This button event is sent by XI to notify your application that the user has pressed a button.
XIE_CELL_REQUEST	XI sends this list event to ask your application to supply it with text and other information to display in a cell. XIE_CELL_REQUEST events follow XIE_GET_* record events.
XIE_CHG_CELL	XI sends this list event to notify your application that the user has changed the contents of a cell.
XIE_CHG_FIELD	XI sends this form event to notify your application that the user has changed the contents of an edit field.
XIE_CLEANUP	XI sends this interface event asking your application to free any data stored in the application data of the interface and to destroy any XVT fonts that are used in the interface.
XIE_COMMAND	XI sends this event to notify your application that the user has selected a menu item.
XIE_DBL_CELL	XI sends this list event to notify your application that the user has double-clicked on a cell.
XIE_DBL_FIELD	XI sends this form event to notify your application that the user has double-clicked on an edit field.
XIE_GET_PERCENT	XI sends this event asking your application to indicate the percentage through the data for a record in the list. XI uses the value returned to set the thumb position on the scroll bar.
XIE_INIT	This interface event is sent to notify your application that the interface has been instantiated and that you should initialize its application data structures.
XIE_REC_ALLOCATE	XI sends this event when your application should allocate space for a record on the list.
XIE_REC_FREE	XI sends this event when your application should free space allocated during an XIE_REC_ALLOCATE event.
XIE_UPDATE	XI sends this event after XI is done drawing the interface. You can draw on top of the XI interface during this event.
XIE_VIR_PAN	XI sends this event when an XI virtual interfacevirtual interface has been panned.

7.4 XI Focus Model

focus modelAs you saw briefly in the last section, XI generates events to notify your application that the user wants to move the focus. The object with the focus is the object currently receiving keyboard input. However, when we talk about focus flow in this manual, we often say that an object “has the focus”, is “losing the focus” or “gains the focus”. Objects that can “have the focus” are edit fields in a form, cells in a list or buttons in a container or on an interface. When an edit field or cell has the focus, the characters the

user types will be displayed in the edit field or cell. When a button has the focus, the user presses the space bar to “press” the button. The interface can have the focus in the sense that no control within that interface has the focus.

There is a special case when you have a “single selection” list. In this case, the focus is on the list object and arrows move the selection within that list. Whenever that selection moves, an **XIE_SELECT** event is generated. The space bar and enter keys will generate **XIE_DBL_CELL** events.

When the user is finished typing in an edit field or cell, and wants to move to another part of the interface, the object is said to be “losing the focus”. If you refuse to let this happen, the object will still “have the focus”. If you allow the object to “lose the focus” and another object to “gain the focus” then the focus will move.

While an object is losing the focus and another object is gaining it, XI will send you **XIE_OFF_*** and **XIE_ON_*** events. As you might expect, **XIE_OFF_*** events are sent to notify you that an object is losing the focus while **XIE_ON_*** events are sent to notify you that an object is gaining it. For example, if a user is tabbing to the next edit field in a form, your event handler might receive an **XIE_OFF_FIELD** followed by an **XIE_ON_FIELD**. If you refuse either of them, the focus doesn’t move.

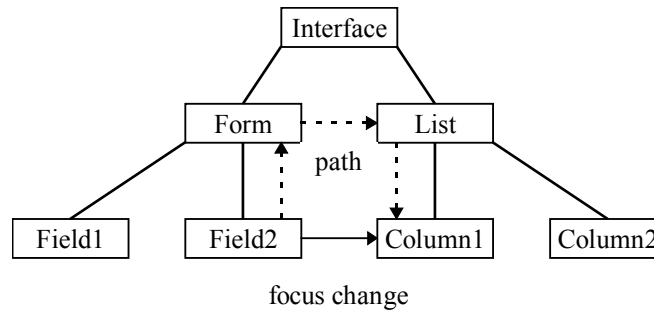
The ability to refuse focus movement is the fundamental reason why XI makes it easier to write database applications. This is because database applications want to verify that the data typed in an edit field meets some criteria before allowing the focus to move to another edit field, and this is exactly what XI simplifies.

Please note that selection of rows, columns, and ranges of cells has nothing whatsoever to do with focus. As we mentioned above, focus refers to keyboard focus.

7.4.1 Basic Focus Rules

There are some basic rules to remember about focus control in XI.

- Rule 1:** **Only one object can have the focus at any given time.**
Two edit fields or cells cannot receive input from the keyboard at the same time.
- Rule 2:** **The object hierarchy determines which events are sent:**
Events are sent corresponding to the objects the focus must encounter to get from one place in the interface to another. As the focus moves from one part of an object hierarchy to another part, events are sent to indicate which objects are losing the focus, and which objects are gaining the focus. For example, if the user moves the focus between two edit fields on the same form, your event handler would get an **XIE_OFF_FIELD** event followed by an **XIE_ON_FIELD** event. However, if the focus moves from an edit field to a cell in a list, then you will get **XIE_OFF_FIELD** and **XIE_OFF_FORM** events followed by **XIE_ON_LIST**, **XIE_ON_COLUMN**, **XIE_ON_ROW**, and **XIE_ON_CELL** events as shown below.



Event Sequence	Object
XIE_OFF_FIELD	Field2
XIE_OFF_FORM	Form
XIE_ON_LIST	List
XIE_ON_COLUMN	Column1
XIE_ON_ROW	Row psuedo-object
XIE_ON_CELL	Cell psuedo-object

Figure 22 - Focus Flow and Object Hierarchy

- Rule 3:** **An application can refuse any focus change at any time.**
 As the user moves the focus from one place in the interface to another, any of the events generated can be refused, and the focus won't move. Contrast this with the way XVT and most native window systems operate. In these systems, you are told that the focus has left and has gone somewhere else. Therefore, your application can't really do anything about it until it has happened. For example, on Microsoft Windows, if the focus is on an edit field and the user clicks on another edit field, you cannot respond until the focus has moved. Then you have to set the focus back to that edit field.
- Rule 4:** **XI interfaces are independent of one another.**
 If the user has two windows up, and has an XI interface in each one, clicking on an inactive window does not generate focus events. We designed XI this way because users will want to browse data in unrelated databases without having to enter valid data before activating other windows. Requiring them to enter valid data before activating another window would be like a word processor requesting that you save your file before you can enter data in your spreadsheet program. This would be contrary to the look and feel GUI system users expect.
- Rule 5:** **Pressing a button doesn't change the focus**
 Because of this, focus events will not be generated upon the button press. Sometimes, this is desired behavior. For instance, you may have a help button on your interface, and when the user presses the help button, your application needs to find out where the focus was when the button was pressed. It can do so by calling **xi_get_focus**. Sometimes, this is not desired behavior. For example, on an OK, or Save Record button, you would like the application to get the **XIE_OFF_*** events, so that it can validate data. In this case, you can force the focus events to take place by moving the focus to the interface by calling the **xi_move_focus** function.

7.5 Event Categories

We have classified XI events into 6 categories: interface events, list events, form events, button events, focus events and special events. There is some overlap between the categories, for example, you will receive focus events while operating a list or form, but our classification is based on the reason why the event is sent, not what object sent it. This is important because it makes a difference in the kinds of information you can get from the event structure as described later.

7.5.1 Interface Events

interface events Interface events are sent to inform you about the life cycle of the interface. These events are **XIE_INIT**, **XIE_CLOSE** and **XIE_CLEANUP**.

After you create an interface, the first event you will receive is an **XIE_INIT**. This event tells your event handler that the interface has been created and it is time to initialize any data structures and do whatever else it needs to do to set things up. Since this event is sent to inform you that the interface was created by XI, you cannot refuse it.

XIE_CLOSE tells the application that the user wants to shut down the interface by clicking the close box. An application can refuse an **XIE_CLOSE** event. This prevents the interface from closing. This event will not occur in the case of calling **xi_delete** on the interface.

XIE_CLEANUP is exactly the opposite of **XI_INIT**. **XIE_CLEANUP** tells you to free your data structures. Techniques for initializing data and freeing it is discussed in detail in the chapter, *Managing Application Data*. See the following table for a summary of interface events.

Interface Events

XIE_INIT	This interface event is sent to notify your application that the interface has been instantiated and that you should initialize its application data structures.
XIE_CLOSE	XI sends this interface event to inform your application that the user attempted to close the window containing the interface. You refuse the event if you want the user to finish doing something else before deleting the interface.
XIE_CLEANUP	XI sends this interface event asking your application to free any data stored in the application data of the interface and to destroy any XVT fonts that are used in the interface.

7.5.2 List Events

list events XI will send your event handler four kinds of list events: record request events, events sent concerning the contents of a cell, events sent indicating that the user is trying to perform some operation on a cell, and miscellaneous list events. Record events are used by XI to help your application manage the record data that corresponds to the text displayed in a list. How to use these events to manage records is explained in detail in the *Using XI Objects* chapter. For now, just concentrate on why the events are sent so that you can compare them to other list events.

As you'll see in the next chapter, XI will send record request events to your application whenever it needs to get records from the database and copy them into memory. (By database, we mean where you store your data. It doesn't have to be a file on the disk). The record request events your event handler can receive are **XIE_GET_NEXT**, **XIE_GET_NEXT**, **XIE_GET_PREV**, **XIE_GET_PREV**, **XIE_GET_FIRST**, **XIE_GET_FIRST**, **XIE_GET_LAST**, **XIE_GET_LAST**, **XIE_CELL_REQUEST**, **XIE_CELL_REQUEST**, **XIE_REC_ALLOCATE**, **XIE_REC_ALLOCATE**, and **XIE_REC_FREE**, **XIE_REC_FREE**. As you will see later in *Using XI Objects*, XI associates record handles with rows in the list so that it can help you manage the memory copies of the records whose text is displayed in the list. When XI wants to display a record it will send your event handler an **XIE_GET_*** event to tell you which record to get from your database. The **XIE_GET_*** events can be refused if there is no record to be read. If you need to allocate information and associate the information with record handles on the list, you can allocate and free the information upon the **XIE_REC_ALLOCATE** and **XIE_REC_FREE** events.

Once a database record has been found and copied into memory, your event handler will return to XI. Then, as XI needs to display actual visible cells, XI will start sending your event handler

XIE_CELL_REQUEST events requesting that it format the record data into text so that XI can display it in the corresponding row. **XIE_CELL_REQUEST** says, “now that you’ve given me a handle to this data record, let me ask you for the text of each cell to be displayed”.

In addition to record request events, there are three other list events XI will send:

XIE_DBL_CELL, **XIE_CHAR_CELL**, and **XIE_CHG_CELL**. Each of these events are sent to notify your application that the user is manipulating a cell. **XIE_DBL_CELL** indicates that the user double clicked on a cell and **XIE_CHG_CELL** indicates that the user has changed the cell text. **XIE_CHAR_CELL** indicates that the user is typing characters into the cell. Since **XIE_DBL_CELL** and **XIE_CHG_CELL** both tell you that something has already happened to the cell, it is meaningless to refuse them.

Other events sent concerning a list are focus events. These are sent to inform you that the user is attempting to move the focus to another place on the interface. List focus events are **XIE_OFF_CELL**, **XIE_ON_CELL**, **XIE_OFF_ROW**, **XIE_ON_ROW**, **XIE_OFF_COLUMN**, **XIE_ON_COLUMN**, **XIE_OFF_GROUP**, **XIE_ON_GROUP**, **XIE_OFF_LIST**, and **XIE_ON_LIST**. All of these are standard focus events explaining where the focus is moving. Focus events are grouped in their own category and are described later. See the following table for a summary of list events.

XIE_ROW_SIZE The **XIE_ROW_SIZE** event is sent when the user dynamically sizes a row. The **XIE_COL_SIZE** event is sent when the user dynamically sizes a column. The **XIE_COL_MOVE** event is sent when the user dynamically moves a column. The **XIE_COL_DELETE** event is sent when the user dynamically deletes a column.

XIE_SELECT **XIE_SELECT** is an event that XI sends when the user selects a row, a column, or a range of cells on the list.

List Events

XIE_CELL_REQUEST	XI sends this cell event to your application asking it to supply XI with text to display in a cell.
XIE_CHAR_CELL	XI sends your application this cell event to notify it that the user is typing into a cell. You can refuse this event to prevent the character from being inserted into the cell.
XIE_CHG_CELL	XI sends your application this cell event to notify it that the user has changed the contents of a cell.
XIE_DROP_ROW	XI sends your application this row event to notify it that the user attempted to move a row by dragging and dropping it. You refuse the event if you do not want the row moved.
XIE_DBL_CELL	XI sends your application this cell event to notify it that the user has double-clicked on a cell.
XIE_GET_FIRST	This record event is sent by XI to ask that your application get the first record displayed in a list. This event is most often seen when initializing a list. You refuse the event if there is no record.
XIE_GET_LAST	This record event is sent by XI to ask that your application get the last record displayed in a list. This event is most often seen when you move the thumb to the bottom of the vertical scroll bar. You refuse the event if there is no record.
XIE_GET_NEXT	This record event is sent by XI to ask your application to get the next record displayed in a list. This event is most often seen when

initializing a list and during a downward scrolling operation. You refuse the event if there is no record.

XIE_GET_PERCENT

XI sends this event asking your application to indicate the percentage through the data for a record in the list. XI uses the value returned to set the thumb position on the scroll bar.

XIE_GET_PREV

XI asks your application to get the previous record displayed in a list. This event is most often seen during an upward scrolling operation. You refuse the event if there is no record.

XIE_OFF_CELL

XI sends your application **XIE_OFF_*** events when notifying it that the user is attempting to move the focus off of an object in the list. These events are summarized in the *Focus Events* section.

XIE_OFF_COLUMN
XIE_OFF_GROUP
XIE_OFF_LIST
XIE_OFF_ROW

See **XIE_OFF_CELL**.

XIE_ON_CELL

XIE_ON_* events are sent to notify your application that the user is attempting to move the focus to an object on the interface. These events are summarized in the *Focus Events* section.

XIE_ON_COLUMN
XIE_ON_GROUP
XIE_ON_LIST
XIE_ON_ROW

See **XIE_ON_CELL**.

XIE_REC_ALLOCATE

XI sends this event when your application should allocate space for a record on the list.

XIE_REC_FREE

XI sends this event when your application should free space allocated during an **XIE_REC_ALLOCATE** event.

XIE_ROW_SIZE

XI sends this event when the user dynamically changes the height of a row. You refuse the event if you want the row height to remain as it was.

XIE_SELECT

XI sends this event to inform your application that the user selected or deselected either a row, a column, or a range of cells on the list. You refuse the event if you want the row or column not to be selected or deselected. When selecting (or deselecting) a range of cells, the **XIE_SELECT** event is not refusable.

XIE_COL_DELETE

XI sends this interface event to inform your application that the user attempted to delete a column by dragging it off of the list. You refuse the event if you do not want the column to be deleted.

XIE_COL_MOVE

XI sends this interface event to inform your application that the user moved a column. You refuse the event if you want the column to remain in the same position.

XIE_COL_SIZE

XI sends this interface event to inform your application that the user sized a column. You refuse the event if you want the column to remain the same width.

7.5.3 Form Events

form events There are fewer events sent for a form than sent for a list. Although XI does have a mechanism for associating application data with a form, it is not the same as the one used for a list. Therefore, the events sent for a form are ones indicating that an edit field is being manipulated by the user and those that tell you that the user is trying to move the focus. The non-refusable edit field manipulation events are: **XIE_DBL_FIELD** indicating a double click on an edit field and **XIE_CHG_FIELD** indicating that the edit field's contents have been edited. Since the purpose of these events is to tell you that something has already happened, it is meaningless to refuse them. The refusable edit field manipulation event is **XIE_CHAR_FIELD**. Refusing this event tells XI to reject the character typed.

The focus events that are sent for a form are **XIE_OFF_FIELD**, **XIE_ON_FIELD**, **XIE_OFF_FORM**, **XIE_ON_FORM**, **XIE_OFF_GROUP**, **XIE_ON_GROUP**. XI focus events are grouped in their own category and are described later.

Form Events

XIE_CHAR_FIELD	XI sends this edit field event to your application when notifying it that the user has typed into the edit field. You can refuse or modify this event to perform character filtering.
XIE_CHG_FIELD	XI sends this edit field event to your application when notifying it that the user has changed the contents of an edit field.
XIE_DBL_FIELD	XI sends this edit field event to your application when notifying it that the user has double-clicked on an edit field.
XIE_OFF_FIELD	XI sends your application XIE_OFF_* events to notify it that the user is attempting to move the focus off of an object. These events are summarized in the <i>Focus Events</i> section.
XIE_OFF_FORM XIE_OFF_GROUP	See XIE_OFF_FIELD .
XIE_ON_FIELD	XI sends your application XIE_ON_* events to notify your application that the user is attempting to move the focus to an object on the interface. You refuse the event if you want the user to finish doing something else on the interface before moving to the object.
XIE_ON_FORM XIE_ON_GROUP	See XIE_ON_FIELD .

7.5.4 Button Events

When the user presses a button on the interface, an **XIE_BUTTON** event is sent. It is not logical to refuse this event because it is informing you that something has happened. Users can press a button in one of two ways. They can tab onto them and press space bar, or they can click on them with the mouse. There are no focus events for buttons.

7.5.5 Focus Events

As previously explained, focus events are sent to inform you that the user is trying to move the focus in a list or form. The focus events your application can receive for a form are **XIE_OFF_FIELD**, **XIE_ON_FIELD**, **XIE_OFF_GROUP**, **XIE_ON_GROUP**, **XIE_OFF_FORM**, **XIE_ON_FORM**. The focus events sent for a list are **XIE_OFF_CELL**, **XIE_ON_CELL**, **XIE_OFF_ROW**, **XIE_ON_ROW**, **XIE_OFF_COLUMN**, **XIE_ON_COLUMN**, **XIE_OFF_GROUP**, **XIE_ON_GROUP**, **XIE_OFF_LIST**, **XIE_ON_LIST**. All of these events can be refused, as in the case of invalid text entry.

Please read the section in this chapter on the *XI Focus Model* for a conceptual explanation of why the events are generated. In addition, much of the *Using XI Objects* chapter is dedicated to explaining the functions you need to call to control focus movement and how to structure your event handlers to respond to focus events. There you will find many code examples of responding to focus events.

Focus Events

XIE_OFF_CELL	This list event is sent to notify your application that the user is attempting to move the focus off of a cell. You refuse the event if the user has not typed valid data.
XIE_OFF_COLUMN	This list event is sent to notify your application that the user is attempting to move the focus off of a column. You refuse the event if you want to restrict the user to editing only the cells in a single column.
XIE_OFF_FIELD	This form event is sent to notify your application that the user is attempting to move the focus off of an edit field. You refuse the event if the user has not typed valid data.
XIE_OFF_FORM	This form event is sent to notify your application that the user is attempting to leave the form and move the focus to another place on the interface. You refuse the event if the record entered by the user is not valid.
XIE_OFF_GROUP	This form or list event is sent to notify your application that the user is attempting to move outside a group of edit fields or columns. You refuse the event if the data entered in the group is not valid as a whole.
XIE_OFF_LIST	This list event is sent to notify your application that the user is attempting to leave the list and move the focus to another place on the interface. You refuse the event if validation of all records in a list fails.
XIE_OFF_ROW	This list event is sent to notify your application that the user is attempting to move the focus off of the row. You refuse the event if the record entered is not valid.
XIE_ON_CELL	XIE_ON_* events are sent to inform your application that the user is attempting to move the focus to an object on the interface. You refuse the event if you want to prevent editing the object.
XIE_ON_COLUMN	See XIE_ON_CELL .
XIE_ON_FIELD	
XIE_ON_FORM	
XIE_ON_GROUP	
XIE_ON_LIST	
XIE_ON_ROW	

7.5.6 Special Events

XI sends five special events: **XIE_XVT_EVENT**, **XIE_XVT_EVENT**, **XIE_XVT_POST_EVENT**, **XIE_XVT_POST_EVENT**, **XIE_COMMAND**, **XIE_COMMAND**, **XIE_UPDATE**, **XIE_UPDATE**, and **XIE_VIR_PAN**, **XIE_VIR_PAN**.

An **XIE_XVT_EVENT** event is sent for every XVT event that occurs and gives the application the opportunity to respond to XVT events if it wants to. For example, you might want to process events for XVT controls that share the window with an XI interface. This event is also refusable. If refused, XI will not process the event. For example, if you enable edit menu items in order to do your own cut, copy, and paste processing, then you should refuse the **E_COMMAND** events for those menu items.

An **XIE_XVT_POST_EVENT** is sent for every XVT event, after XI has processed the event. This is used in special cases where you may have to wait for XI to finish processing before taking some action. For example, XI may have the mouse trapped, so you will want to wait until after the **E_MOUSE_UP** event is processed by XI and then open an XVT modal dialog.

An **XIE_COMMAND** event indicates that the user chose a menu item. When your application receives an **XIE_COMMAND** event, it will need to look at the menu tag to see what operation the user is attempting.

An **XIE_UPDATE** event is sent upon an **E_UPDATE** event, but after XI has finished drawing the interface. You can draw on top of any XI objects at this point.

XI sends the **XIE_VIR_PAN** event whenever the user pans an XI virtual interface. If you need to move some XI objects in the virtual interface to keep the objects in sync with the XI interface, you will need to write code to respond to this event.

7.6 The XI_EVENT Structure

The data found in the **XI_EVENT** structure is dependent on the type of event sent to your event handler. Since most of the XI events are sent to notify your application that the user is manipulating an object or trying to move the focus, your event handler is usually interested in the part of the **XI_EVENT** structure that gives it information about the object for which the event was sent. Object information is found in the **xi_obj** field.

With the record retrieval events sent for a list object, your application will use different fields in the **XI_EVENT** structure. In particular, it will use the **rec_request** structure inside the event to find out which record it needs to retrieve from the database.

For the events that are sent asking you to format data into text to be displayed in a cell, you would use the **cell_request** structure in the **XI_EVENT** structure.

For events sent indicating that a XVT event was generated, you would use the **xvte** member of the **XI_EVENT** structure to find out if the event is important to you or not.

In the following table you will see a list of XI events, a small description of why each event is sent, and which pieces of the **XI_EVENT** structure each event uses.

Event	Description	Part of XI_EVENT used
XIE_BUTTON	Button press.	xi_obj
XIE_CELL_REQUEST	Request for text for a cell in a list.	cell_request
XIE_CHAR_CELL	User is typing into a cell.	chr
XIE_CHAR_FIELD	User is typing into an edit field.	chr
XIE_CHG_CELL	Contents of cell has changed.	xi_obj
XIE_CHG_FIELD	Contents of an edit field has changed.	xi_obj
XIE_CLEANUP	Free application data structures.	xi_obj
XIE_CLOSE	User pressed the close box to delete the interface	xi_obj
XIE_COL_DELETE	User has deleted a column.	column
XIE_COL_MOVE	User has moved a column.	column
XIE_COL_SIZE	User has resized a column.	column
XIE_COMMAND	User has selected a menu item.	cmd
XIE_DBL_CELL	User has double-clicked on a cell.	xi_obj
XIE_DBL_FIELD	User has double-clicked on an edit field.	xi_obj

Event	Description	Part of XI_EVENT used
XIE_GET_FIRST	Get first record displayed in a list.	rec_request
XIE_GET_LAST	Get last record displayed in a list.	rec_request
XIE_GET_NEXT	Get next record displayed in a list.	rec_request
XIE_GET_PERCENT	Get percentage through data displayed in a list	get_percent
XIE_GET_PREV	Get previous record displayed in a list.	rec_request
XIE_INIT	Initialize the interface.	xi_obj
XIE_OFF_CELL	Focus change event.	xi_obj
XIE_OFF_COLUMN	Focus change event.	xi_obj
XIE_OFF_FIELD	Focus change event.	xi_obj
XIE_OFF_FORM	Focus change event.	xi_obj
XIE_OFF_GROUP	Focus change event.	xi_obj
XIE_OFF_LIST	Focus change event.	xi_obj
XIE_OFF_ROW	Focus change event.	xi_obj
XIE_ON_CELL	Focus change event.	xi_obj
XIE_ON_COLUMN	Focus change event.	xi_obj
XIE_ON_FIELD	Focus change event.	xi_obj
XIE_ON_FORM	Focus change event.	xi_obj
XIE_ON_GROUP	Focus change event.	xi_obj
XIE_ON_LIST	Focus change event.	xi_obj
XIE_ON_ROW	Focus change event.	xi_obj
XIE_REC_ALLOCATE	Allocate a record handle.	rec_allocate
XIE_REC_FREE	Free a record handle.	rec_free
XIE_ROW_SIZE	User has resized a row.	row_size
XIE_SELECT	User has selected an object.	select
XIE_UPDATE	A region has been invalidated.	xvte
XIE_VIR_PAN	A virtual interface has been panned.	vir_pan
XIE_XVT_EVENT	XVT event before XI has processed it.	xvte
XIE_XVT_POST_EVENT	XVT event after XI has processed it.	xvte

8

Using XI Objects

Using XI Objects is perhaps the most practical chapter in the manual. In this chapter you will find code examples for using all of the objects found within an XI interface such as edit fields, forms, lists, cells, rows, columns, groups, buttons and static text. Within each section, you will find code examples of responding to events generated when the object is used. In addition, you will find out how to interface with a database, when to validate data, and how to change the object's look and feel.

getting an object pointer **8.1 Getting an Object Pointer**

Throughout this chapter you will be using XI functions that take an object pointer. Because of this, you will need to be able to get a pointer to an object if you don't have one. There are several ways to do this as outlined below.

getting an object from an event **8.1.1 Getting an Object from an Event Structure**

Most of the time you will get the object you need from an event sent by XI notifying you that something has happened to the object. For example, when the user tries to move the focus off of an edit field, your event handler will receive an `XIE_OFF_FIELD` event. Inside the `XI_EVENT` structure sent with the event, you will receive an object pointer to the affected edit field, which you can then use to manipulate the edit field in some fashion such as calling `xi_get_text` to get the edit field's text.

The following code, from "lstlink.c", demonstrates getting an object pointer from the event structure.

```

case XIE_CHAR_CELL:
{
    XI_OBJ*    cell = xiev->v.chr.xi_obj;
    LINK_FIELD field = column_to_field( cell->parent,
                                       cell->v.cell.column );

    if ( field == LINK_DATE )
        xiev->refused = !validate_date_char( xiev->v.chr.ch );
    break;
}

```

8.1.2 Using a Control ID

Recall that each object in the form has a unique control ID. The ability to assign a control ID to an object has several advantages. First of all, you can know the control ID of an object at compile time, but you cannot know the address of an object until it has been instantiated. Second, using an object's control ID is more convenient for such things as locating an object in a tree of objects (such as an interface hierarchy) or setting up a switch statement to act on the control ID. Both of these methods of identifying objects are interchangeable—you can always get an object's control ID if you have its pointer, and you can get the pointer to the object if you have its control ID. This gives you a great deal of flexibility in the programming techniques you can use to manipulate objects.

If you know the control ID for an object and have the pointer to the interface object for the window the object is in, then all you have to do is call `xi_get_obj` passing it the interface object pointer and control ID. Remember that `xi_create` will hand you a pointer to the interface object when you instantiate the interface and the interface pointer is passed to your event handler function.

The following code, from "lstmem.c", demonstrates the use of `xi_get_obj` to get an object pointer.

```

case M_SELECT_NONE:
    if ( xi_move_focus( itf ) )
    {
        select_clear( itf, TRUE, TRUE );
        xi_cell_request( xi_get_obj( itf, LIST_CID ) );
    }
    break;

```

children of objects8.1.3 Getting an Object's Children

When you have a pointer to the parent of the object you want to get, you can call `xi_get_member_list` to get the object you need. `xi_get_member_list` returns an array of object pointers to the children of the object you passed to it. For example, if you have an interface, and would like to know what children the interface has, you would pass the object pointer of the interface to `xi_get_member_list`, and it will tell you which lists, forms, containers, buttons, rectangles, lines, and static text are in the interface. You can also use `xi_get_member_list` to start at the top of an interface and walk the entire interface tree down to the edit fields, columns and buttons at the bottom.

The following code, from "lstmem.c", demonstrates the use of the object's children.

```

static void select_set_color( XI_OBJ* itf, COLOR color )
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data(itf);
    REC_INFO* rec = list_info->records;
    int num, col_num, count;
    XI_OBJ* list = xi_get_obj( itf, LIST_CID );
    XI_OBJ** column;

    for ( num = 0; num < list_info->nbr_records; num++, rec++ )
    {
        mem_select_color( rec, color );
        column = xi_get_member_list( list, &count );
        for ( col_num = 0; col_num < count; col_num++, column++ )
            if ( ( xi_get_attrib( *column ) & XI_ATR_SELECTED ) != 0 )
                mem_set_color( rec, (*column)->cid - COL_BASE_CID, color );
    }
    xi_cell_request( list );
}

```

parent of objects 8.1.4 Getting the Parent of an Object

Any object that is a child of another has access to the parent through the **parent** member of the **XI_OBJ** structure. This means that a column or cell can easily get the object pointer for the list that it belongs to using this pointer.

The following code, from “lstdb.c”, demonstrates the use of the **parent** pointer.

```

case XIE_OFF_GROUP:
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );
    XI_OBJ* cell = xi_get_focus( itf );

    if ( list_info->row_changed && !emp_validate_hours(
        row_to_record( cell->parent, cell->v.cell.row ) ) )
    {
        xvt_error( "Minimum hours cannot be greater than maximum." );
        xiev->refused = TRUE;
    }
    break;
}

```

8.1.5 Getting the Object With Focus

At any time that some object has the focus, you can get a pointer to that object by calling **xi_get_focus**. The code example in the previous section demonstrates the use of this function.

8.1.6 Making a Pseudo-Object

The last approach is manufacturing pseudo-objects for cells and rows. Since these objects don’t appear in the interface hierarchy directly and are never instantiated, you will sometimes need to manufacture them for the purpose of passing an object pointer to the functions that take object pointers. In this case, you would use the **XI_MAKE_CELL**, **XI_MAKE_CELL** and **XI_MAKE_ROW**, **XI_MAKE_ROW** macros to manufacture pseudo objects to describe these cells and rows.

The following code, from “lstlink.c”, demonstrates the manufacturing of a pseudo-row.

```

case DELETE_CUR_CID:
{
    XI_OBJ* obj = xi_get_focus( itf );

    if ( obj->type == XIT_CELL )
    {
        XI_OBJ row;

        link_delete( row_to_record( obj->parent, obj->v.cell.row ) );
        XI_MAKE_ROW( &row, obj->parent, obj->v.cell.row );
        xi_delete_row( &row );
        update_numbers( obj->parent );
    }
    break;
}
}

```

8.2 Using Edit Fields

edit fields When using XI edit fields, it is up to you to format your data into text, and put the text into the edit fields. Once your data is formatted correctly, you call `xi_set_text` to set the text of the edit field. Please note that XI uses a different mechanism to handle data in a list as is discussed in the *Using Lists* section.

8.2.1 Being Notified of Typing in an Edit Field

Your application will be given a chance to refuse or alter any typing into the edit field during an **XIE_CHAR_FIELD** event. This event is sent whenever characters are typed into an edit field. Your application can refuse the event to disallow certain characters, or modify the event to force certain behavior such as all uppercase or lowercase. If an **XIE_CHAR_FIELD** event is refused, then no change will be made, and no following **XIE_CHG_FIELD** event will be sent.

If the edit field is changed by typing or clipboard activity, XI will send your event handler function an **XIE_CHG_FIELD** event. The purpose of this event is to allow your application to make note of the change so that it knows to verify the edit field contents when an **XIE_OFF_FIELD** event is later received.

8.2.2 Filtering characters

filtering characters The **XIE_CHAR_FIELD** event is most often used to restrict the kinds of characters the user can type into an edit field, such as only digits for a zip code, letters for a name, digits separated by ‘(’, ‘)’ and ‘-’ for a phone number, etc.

To limit the kinds of characters the user can type, you need to respond to the **XIE_CHAR_FIELD** event by altering or refusing the character sent to your event handler. To display only the allowable characters in the edit field, set the `xiev->refused` flag to **TRUE** whenever an illegal character is passed in the `xiev->v.chr.ch` field. When validating characters, you should ignore any characters that are not ascii or not printable. These characters are editing characters, like left and right arrow and backspace.

The following code, from “`lstlink.c`”, demonstrates validation of characters for a date.

```

static BOOLEAN validate_date_char( int ch )
{
    if ( ch > 255 || !isprint( ch ) )
        return TRUE;
    if ( ch == '/' || ch == '-' || isdigit( ch ) )
        return TRUE;
    return FALSE;
}

```

```

static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
    FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

    switch ( xiev->type )
    {
        ....
        case XIE_CHAR_FIELD:
            if ( xiev->v.chr.xi_obj->cid == FIELD_BASE_CID + LINK_DATE )
                xiev->refused = !validate_date_char( xiev->v.chr.ch );
            break;
        ...
    }
}

```

When processing an **XIE_CHAR_FIELD** event, it is also possible to change characters as they are typed by changing the **xiev->v.chr.ch** field. In this fashion, for example, you could make an edit field where all characters are converted to upper case.

8.2.3 Validating Edit Field Text

validating fields When the user is finished editing an edit field and attempts to move off of the edit field, you'll want to examine the contents of the edit field to verify that the user entered something reasonable before letting the focus move. For example, the user may be entering data for a key field such as a part number or social security number. In this case, you will want to check the database to make sure that the user entered a valid database key before letting him continue. You could also check for valid dates, currency amounts and other kinds of data.

The basic idea here is that you let users freely edit the contents of an edit field until they try to move the focus to another object in the interface by pressing the TAB key or clicking the mouse on another edit field or cell. When the user tries to change the focus to a different object in the window, your event handler will be notified of this by an **XIE_OFF_FIELD** event. When you receive this event you have the opportunity to refuse the event if the contents of the edit field are invalid.

It is important to note that you do not receive an **XIE_OFF_FIELD** event when the user switches to another window. This is because the user may want to switch to another window to browse some other data. XI makes sure that such switching does not force a focus event to occur.

8.2.3.1 Checking for Valid Data

Whenever your event handler receives an **XIE_OFF_FIELD**, your application can check that valid data was entered before letting the user move the focus by calling **xi_get_text** on the edit field, and then handing the text returned by **xi_get_text** to a validation routine.

If your validation routine rejects the data entered by the user, then you'll want to refuse the **XIE_OFF_FIELD** event by setting the **refused** member of the **XI_EVENT** structure to **TRUE**. This will tell XI not to move the focus after all. To let the user know that something went wrong, you may also want to beep by calling the XVT function **xvt_beep**, and then select the text by calling **xi_set_sel**. This will result in highlighting the text so that when the user starts typing, the previous text will be replaced with the new text.

The following code, from "lstdb.c", demonstrates validation of the field. The actual validation is performed in **emp_set_value** from "datdb.c".


```

case XIE_OFF_FIELD:
    if ( form_info->field_changed )
    {
        XI_OBJ* field = xiev->v.xi_obj;

        if ( !emp_set_value( form_info->handle, field->cid -
FIELD_BASE_CID,
                        xi_get_text( field, NULL, 0 ) ) )
            xiev->refused = TRUE;
        else
        {
            form_info->changed = TRUE;
            form_info->field_changed = FALSE;
            set_field_text( field, form_info->handle );
        }
    }
    break;

```

8.2.4 Changing Edit Field Attributes

attributes, fields field attributes You can dynamically change the look and feel of an edit field by calling `xi_set_attrib` with a pointer to the edit field object and a bitwise OR'ed combination of values corresponding to the new attributes you want the edit field to have. Keep in mind that you cannot change the attributes of an edit field while the edit field has the focus. Therefore, before you can call `xi_set_attrib` on an edit field that has the focus, you will need to move the focus to the interface by calling either `xi_set_focus` or `xi_move_focus` with the interface object. Note that moving the focus to the interface merely removes the keyboard focus from any objects in the interface. This means that XI generates all of the appropriate `XIE_OFF_*` events, but does not generate any `XIE_ON_*` events.

As explained in *Characteristics of XI Objects*, the attributes you can set for an edit field are : `XI_ATR_ENABLED`, `XI_ATR_VISIBLE`, `XI_ATR_EDITMENU`, `XI_ATR_AUTOSCROLL`, `XI_ATR_AUTOSELECT`, `XI_ATR_RJUST`, `XI_ATR_READONLY`, `XI_ATR_PASSWORD`, `XI_ATR_FOCUSBORDER`. If you want to know what attributes an edit field has, you call `xi_get_attrib`.

8.2.5 Changing a Single Attribute

In most cases, you will want to change only one attribute of the edit field such as disabling the edit field or making it invisible. You would call `xi_get_attrib` to get the attributes for the edit field, change the bit and then call `xi_set_attrib` to give the edit field its new look. The following code segment demonstrates setting the enabled attribute:

The following code, from "lstlink.c", shows how to set or clear the visible attribute based on the boolean value passed to the function.

```

static void change_section( SECTION_INFO* section, BOOLEAN flag )
{
    int      num;
    XI_OBJ** obj;

    for ( num = 0, obj = section->objs; num < section->count;
          num++, obj++ )
        if ( flag )
            xi_set_attrib( *obj, xi_get_attrib( *obj ) | XI_ATR_VISIBLE );
        else
            xi_set_attrib( *obj, xi_get_attrib( *obj ) & ~XI_ATR_VISIBLE );
}

```

8.3 Using Forms

forms, usingSince forms are composed of edit fields, you will need to know how to manipulate edit field objects before you can manipulate form objects. Methods you'll need to use to manipulate edit fields were the topic in the last section. There you learned how to call `xi_set_text` to set the text of an edit field, filter characters the user types, check that the user has entered valid data and change an edit field's attributes. In this section, you will learn how to associate record data with a form and check that the user has entered a valid record. In addition, you will learn how to set the navigation sequence of the form.

8.3.1 Validating the Contents of a Form

forms, validating contentsAs described in the *Using Edit Fields* section, you can validate the contents of each edit field in a form as the user tries to move the focus off of each of them. However, it may be your chosen look and feel to delay validation until the whole form has been entered. For example, in a database application running on a network, you might choose to wait until the user performs an action such as clicking on an "Add Record" button before checking with the server that the data entered in the form is valid.

Regardless of the reason for delaying validation, the way that you would begin the process of validating the entire contents of a form is to get the form object by calling `xi_get_obj` passing it the interface pointer and control ID of the form (or during an `XIE_OFF_FORM` event, you can pass `xienv->v.xi_obj`). Once you have the object pointer, you would then call `xi_get_member_list` to retrieve the list of edit fields contained in that form. After you have the list of children edit fields, you can loop through the list and retrieve the text for each object.

8.3.2 Interfacing to Databases When Using a Form

Most of the time, an XI form will display only one record at a time and you will want to keep a copy of the database record currently being displayed in memory. Also, you will want to write the changes made by the user out to the database after you have checked that those changes are valid. Because of this, there are two issues you must address when interfacing to a database. You must figure out when to update the memory copy of your record and when to store the updated record in the database. In the following discussion, we describe the opportunities XI gives you to read a record from the database, verify that text the user has entered is valid, and write the record to the database. Keep in mind that our suggestions for interfacing to the database are only hints, and may not be appropriate for your application.

The following diagram may help in illustrating the techniques described in the following sections.

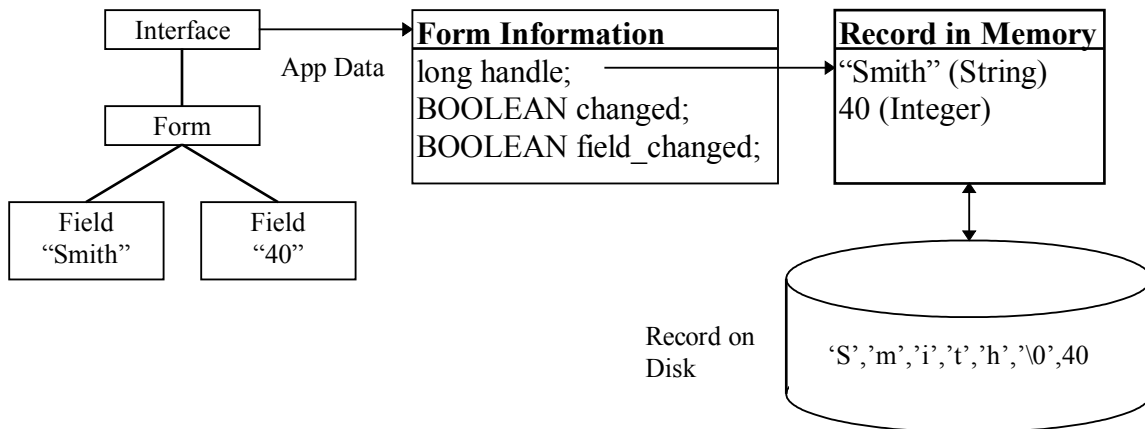


Figure 23 - Editing a Database Record in a Form

8.3.2.1 Reading a Database Record

In most cases, your form will display text for a single database record. Unlike lists, where there are many record handles to be managed by your application and XI, forms are quite simple. You need to keep only one record in memory for the entire form.

Although you can set application data for the form object, it is usually easier to set the application data for the interface. This would only be a problem in more complex interfaces. For example, if an interface had both a form and a list in it.

The following code, from “lstdb.c”, shows how the application data is initialized for the “change” form. The **handle** is used to refer to the database record in memory.

```
static void open_chg_form( XI_OBJ* list_itf, long handle )
{
    FORM_INFO* form_info;
    XI_OBJ*     itf;

    itf = create_emp_form( FALSE );
    form_info = (FORM_INFO*)xi_tree_malloc( sizeof( FORM_INFO ), itf );
    form_info->list_itf = list_itf;
    form_info->handle = handle;
    form_info->allocated = FALSE;
    xi_set_app_data( itf, PTR_LONG( form_info ) );
    xi_dequeue();
}
```

During the **XIE_INIT** event, we need to set the text for the edit fields to match that of the database record we are editing. The following code, also from “lstdb.c”, demonstrates this.

```
case XIE_INIT:
{
    int      num;
    XI_OBJ*  form;
    XI_OBJ** field;

    form = xi_get_obj( itf, FORM_CID );
    field = form->children;
    for ( num = 0; num < form->nbr_children; num++, field++ )
        set_field_text( *field, form_info->handle );
    break;
}
```

8.3.2.2 Marking a Record as Edited

As the user makes changes to text, you will need to track those changes so that you can make sure that the changes are valid and write the record to the database. We recommend you do this by allocating two extra **BOOLEAN** flags associated with form. These flags will be used to track the status of the record as the user makes changes. The first flag keeps track of whether or not the edit field that has focus has changed. The second flag keeps track of whether or not any data has been changed for the record.

The purpose of the first flag, **field_changed**, is to prevent extra validations and to allow the second flag, **changed**, to be set accurately. In other words, we have to know if the edit field was changed to know if the record was changed. The **changed** flag serves two purposes. It is used to determine whether or not the record actually needs to be updated when the user hits the “Save” button and it allows us to prompt the user if they try to cancel the window.

The following code, from “lstdb.c”, demonstrates the use of these flags.

```

static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
    FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

    switch ( xiev->type )
    {
        ...
        case XIE_CLOSE:
            if ( ( !xi_move_focus( itf ) || form_info->changed )
                && xvt_ask( "No", "Yes", NULL,
                    "You have made changes. Are you sure you want to cancel?" )
                    != RESP_2 )
                xiev->refused = TRUE;
            break;
        case XIE_BUTTON:
            switch ( xiev->v.xi_obj->cid )
            {
                case CHG_BTN_CID:
                    if ( !xi_move_focus( itf ) )
                        break;
                    if ( form_info->changed )
                        emp_update( form_info->handle );
                    refresh_row( xi_get_obj( form_info->list_itf, LIST_CID ),
                        form_info->handle );
                    form_info->changed = FALSE;
                    xi_delete( itf );
                    break;
                case CANCEL_BTN_CID:
                    if ( form_info->changed && xvt_ask( "No", "Yes", NULL,
                        "You have made changes. Are you sure you want to
cancel?" )
                            != RESP_2 )
                        break;
                    xi_delete( itf );
                    break;
            }
            break;
        case XIE_CHG_FIELD:
            form_info->field_changed = TRUE;
            break;
        case XIE_OFF_FIELD:
            if ( form_info->field_changed )
            {
                XI_OBJ* field = xiev->v.xi_obj;

                if ( !emp_set_value( form_info->handle,
                    field->cid - FIELD_BASE_CID,
                    xi_get_text( field, NULL, 0 ) ) )
                {
                    xiev->refused = TRUE;
                    xi_set_sel( field, 0, INT_MAX );
                }
                else
                {
                    form_info->changed = TRUE;
                    form_info->field_changed = FALSE;
                    set_field_text( field, form_info->handle );
                }
            }
            break;
    }
}

```

8.3.2.3 Validating Edit Field Text

It isn't until your application receives an `XIE_OFF_FIELD` event that it should validate text entered by the user, and update the record stored in memory if the data is valid. If the text is not valid, your application should refuse the `XIE_OFF_FIELD` event. Upon receiving the event you could either reset the text in the edit field to what it was before the user made changes or select the text in the edit field showing him that he must try again.

See the code above for an example of this.

8.3.2.4 Marking a Record as Modified

The second flag indicates that the record stored in memory has been modified and no longer matches the record in the database. You should set this flag to true only after you have verified the text upon receiving an `XIE_OFF_FIELD` for a record marked as edited, and have updated the copy of the record kept in memory. Since the text displayed on the screen now reflects the data stored in memory, you will also need to reset the edited flag to false indicating its new state.

See the code above for an example of this.

8.3.2.5 Writing Database Records

An application can write modified records to the database in two different ways when using a form. Let's call these approaches "automatic" and "manual".

Writing records automatically: The automatic method implies that records will be written to the database immediately before a different record will be displayed on the form. This will have a look-and-feel of allowing users to freely edit database records and have changes made to them stored automatically without requiring them to press a "Save Record" button.

To use this approach, your application will need to write a modified record before another record is read and displayed in the form. Since XI does not automatically read new records into the form for your application, you will need to have a menu item or button called something like "Next Record" or "Previous Record". When the user presses this button, then your event handler will receive an `XIE_BUTTON` event, and should write the record to the database if it has been marked as modified. Then it can read the new record and display its text in the form.

Writing records manually: You can use the manual method if your application has a button or menu item that is there for the sole purpose of writing a modified record to the database. This button might be labelled "Save". When the user presses this button, your application will receive an `XIE_BUTTON` event. Upon receiving this event, your application should move the focus to the interface, and then write the record to the database if the modified flag for that record has been set.

With either method, after writing the record, your application should reset the modified flag to `FALSE` to indicate that the memory copy of the record contains the same data as the one in the database.

See the code above for an example of writing records manually.

8.3.3 Setting the Keyboard Navigation Sequence

As the user tabs from edit field to edit field on the form, the focus will move in a characteristic navigation pattern. Once you have created the form, it is possible to change the keyboard navigation sequence. The tab control ID can be found in edit fields by referencing `xi_obj->v.field->tab_cid`. It is necessary to define `XI_INTERNAL` before including XI.H if you intend to change the tabbing sequence. Be careful to not set up an invalid tabbing sequence.

When you built an object definition tree you probably used the convenience functions provided by XI. When you added edit field definitions to a form definition, you called the function,

`xi_add_field_def` `xi_add_field_def` passing it the tab control ID for each edit field that you created. When the user presses TAB, the `tab_cid` argument tells XI to move the focus to the edit field whose `cid` is equal to `tab_cid`.

8.4 Using Lists

lists, using Like XI forms, XI lists do not contain the data that your application is using. They only display and allow input of text. Therefore, your application must store the data, format it into text whenever a list asks for it, and convert the text entered by the user into data again. Thus, when using a list there is a division of responsibilities between your application and XI. XI is responsible for displaying text. You are responsible for providing it with text to display.

8.4.1 Record Handles

The way your application and XI exchange information is through data records associated with each row the list can display. Because the list needs to refer to data kept by the application without knowing what the data is, your application hands it a record handle for each record displayed in the list.

The term “record handle” refers to a programming technique where a module refers to data in another module without knowing what the data is. In our case, the record handles are meaningful to your application—they identify records. To XI, they are incomprehensible and XI cannot interpret them in any way. All it can do is hand them back to your application. The record handle you hand XI is of type **long**. Once again, the list has no idea what it means. To the list, it is just an identifier that the list is going to give back to your application.

To see this situation from XI’s perspective, think of a record handle as if it were a theatre ticket. Imagine that a tourist goes to a foreign country and buys a theater ticket. Since he doesn’t speak the language, he cannot understand the text printed on the ticket. However, he doesn’t worry about it because there is a person at the door who can read the ticket and let him see the show. In a similar way, XI will use record handles to ask your application to retrieve and format records.

The following diagram illustrates the relationships between your application, record handles, and XI.

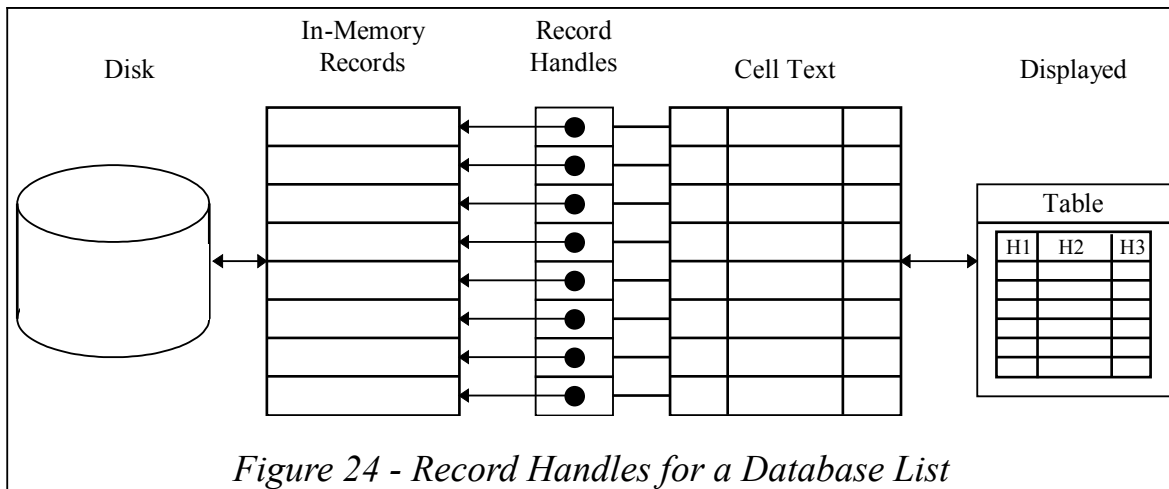


Figure 24 - Record Handles for a Database List

XI knows nothing about the in-memory records. It just stores handles to those records. It gets those handles from your application via the record request events. The record request events are **XIE_GET_FIRST**, **XIE_GET_LAST**, **XIE_GET_NEXT**, and **XIE_GET_PREV**.

The record handle is also passed to your application in the **XIE_CELL_REQUEST** event. The cell text that you return is stored in the XI cell text array, as shown in the diagram above.

The cell text array is internal to XI. You never need to worry about it. You should realize, though, that XI can update the window from the cell text array without generating cell request events (for example, when a popup goes away, and the window needs to be redrawn.)

8.4.2 How XI Manages the Record Handle Array

There are two basic ways that developers wish to use lists:

1. Sometimes developers have list that has relatively few records and requires no editing. For example, they may have a small database table that is used in a drop down list. The values are never changed because it's only used to select one of the records. In this case it is desirable for the XI list to request all records before putting up the XI list. XI will do record request events for all records in the list. However, the user would only be able to see the visible portion of the list. To enable this behavior, set **list_def->v.list->get_all_records** to **TRUE** after calling **xi_add_list_def**. Note that you are limited to 255 records in this kind of list.
2. More commonly, there are lot of records and they may added to or deleted from the list by the user. Having all the records in memory would take up too much memory. It also would probably take to long to read all of the records before displaying the visible portion of the list. In this case, XI will request records as they become visible and then request the cell data in order to display each row of the list. As the list scrolls, records that are no longer visible are discarded, while new records are requested and then displayed. This is the default behavior of XI lists.

8.4.3 Managing Records

Once the application has given XI a set of record handles for the records to be used in the list, then XI will manage those records by associating them with rows in the list. If your application needs to allocate space for the records, it is often convenient to have the record handles be pointers to the allocated memory, cast into type **long**. Therefore, you may want to respond to the **XIE_REC_ALLOCATE**/**XIE_REC_ALLOCATE** event to allocate space for your record handle. If you do so, you must respond to **XIE_REC_FREE** events to free that allocated space. Do not use tree memory for these allocations.

In the next few pages, you will find a description of the events XI uses to manage records, and the code you'll need to write in order to respond to those events. You will discover what you'll need to do when initializing the list, retrieving records, filling them with data, processing user input and writing records to the database. Keep in mind that throughout this discussion we will be using the term "database" to mean "the place your application stores data". We use this term in the general sense—please don't think that your data must be stored in a file on the disk. It could be stored in any format on any medium. The only requirement is that your application knows the format of the records, can associate a handle with each record, and can give XI text to display when XI requests it.

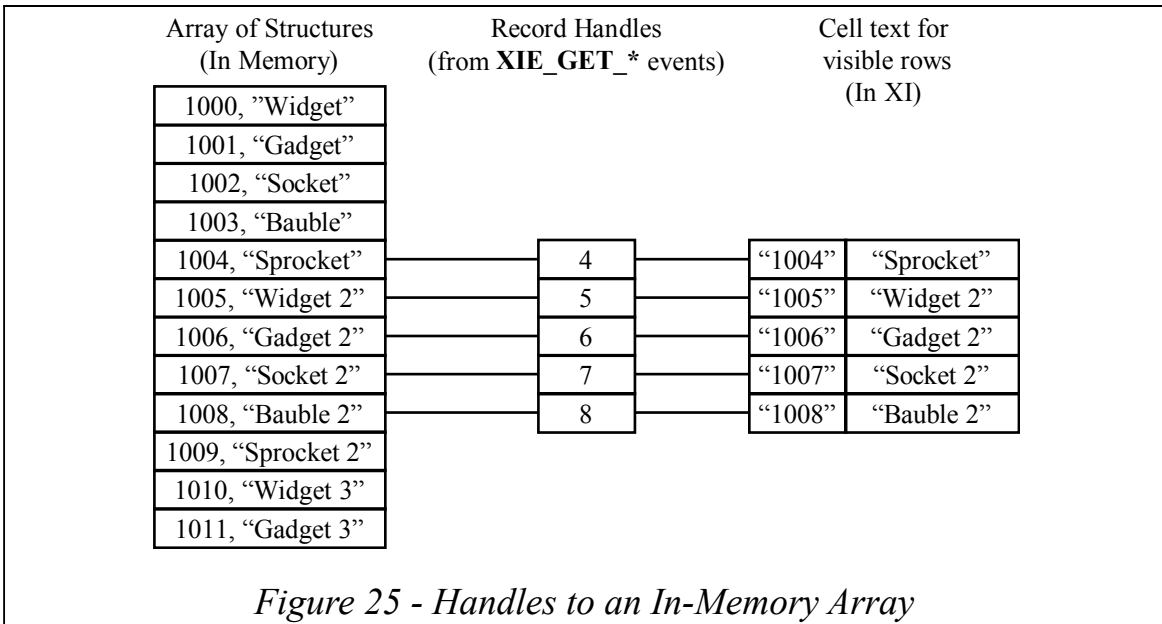
8.4.3.1 Associating Record Handles with Data

Because a record handle can be any type of information stored in a **long**, there are many ways you can associate record handles with memory copies of records. There are three main methods of using record handles:

1. The record handles are subscripts to an in-memory array of data. This is the method used for "lstm.c" and "datmem.c".
2. The record handles are pointers to an in-memory structure (e.g. a linked list). This is the method used for "lstlink.c" and "datlink.c".
3. The record handles are allocated structures that refer to records in a file or database table. In this case, the allocated structure must contain the necessary information to access that record for updates, next and previous operations. This is the method used for "lstdb.c" and "datdbc.".

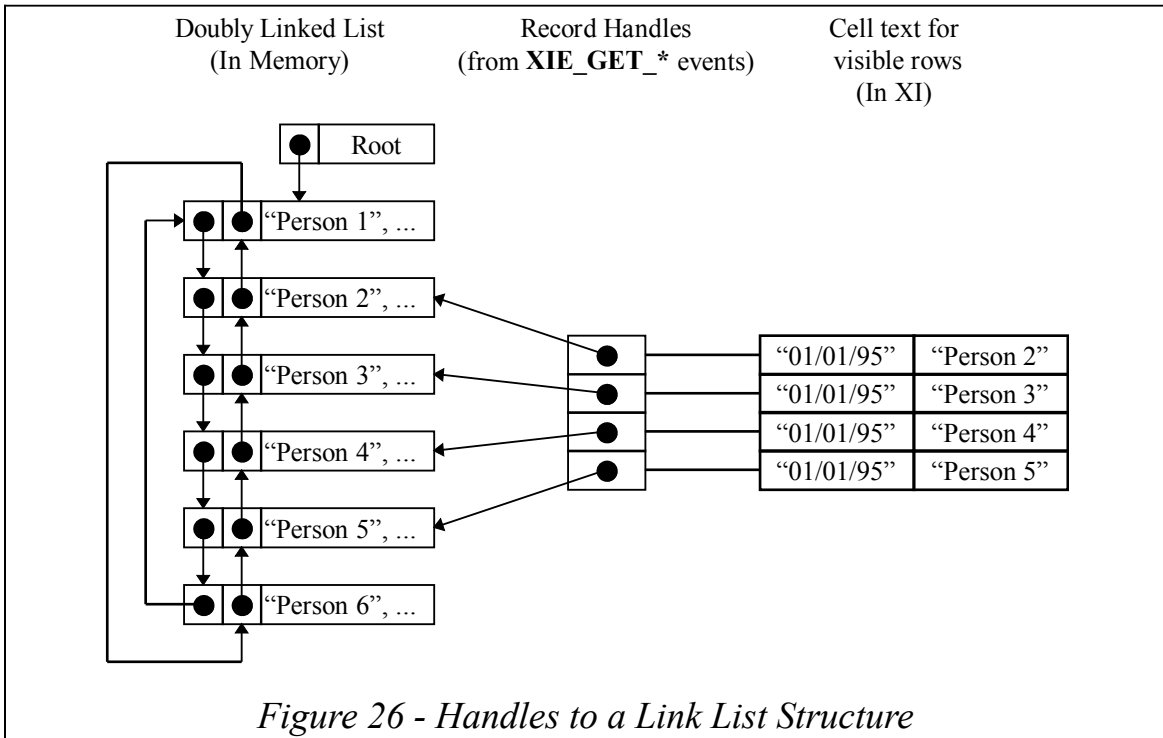
All three of these techniques are demonstrated in the example program.

The following diagram illustrates using array subscripts as your record handles.



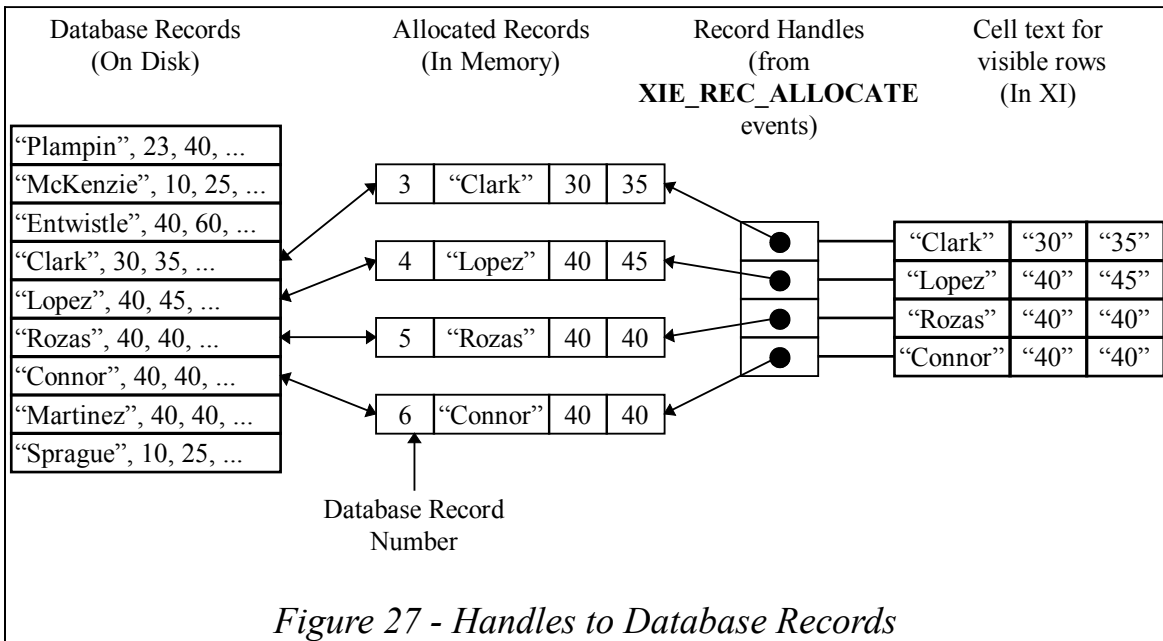
If you have your entire database stored in memory, you would want to refer to your records by record number. Since you don't need to retrieve records from the disk and copy them into memory in response to user activities, you will not need to allocate space for records in response to an **XIE_REC_ALLOCATE** event.

If you are using a linked list structure, you can use the pointers to the "nodes" as the record handles. This allows you to easily respond to all the XI events for the list. The following diagram illustrates this method.



Since the entire linked list resides in memory, you still don't need to allocate memory for the handles. However, the handles are actually pointers rather than integer subscripts.

When the list records come from a large database, you will probably want to avoid reading all of the records into memory. In this case, you can respond to the **XIE_REC_ALLOCATE** event which will allocate a structure to hold information about a particular row in the list. Whenever rows scroll off the list, XI will generate an **XIE_REC_FREE** event to free up that space. In this way, a minimal amount of memory is used to keep track of the current position in the list.



Your application will receive an **XIE_REC_ALLOCATE** event before every **XIE_GET_FIRSTXIE_GET_FIRST**, **XIE_GET_NEXTXIE_GET_NEXT**, **XIE_GET_PREVXIE_GET_PREV**, or **XIE_GET_LASTXIE_GET_LAST** event. During processing of

this event, you have the option of allocating space for the record. In addition, as records are scrolled off of the list, your application will receive **XIE_REC_FREE** events. If you are allocating space for records during processing of the **XIE_REC_ALLOCATE** event, you must free this space during the **XIE_REC_FREE** event.

8.4.3.2 Retrieving Records

Your application will need to respond to the record events XI generates. Keep in mind that unlike other windowing systems, XI will send list events in a specific order, and the order in which you receive these events is important. You can expect list events to come in the following order: **XIE_GET_*XIE_GET_*** events asking you to get data from the database, **XIE_CELL_REQUESTXIE_CELL_REQUEST** events asking you to put text into cells, **XIE_ON_*XIE_ON_*** events sent to notify you that the focus has entered a cell, **XIE_CHAR_CELL** events sent to notify you that the user has typed into a cell, and **XIE_OFF_*XIE_OFF_*** events sent to notify you that the focus has left a cell. Therefore, you'll never be asked to put text into cells before you have the data record, or notified that the contents of the cell has been changed before you've put text into the cell.

The one thing that is not guaranteed is that you will get the **XIE_CELL_REQUEST** events immediately after the **XIE_GET_*** event for that row. Other **XIE_GET_*** events may be sent, for other rows before you get the **XIE_CELL_REQUEST** events. However, you will never get an **XIE_CELL_REQUEST** for a row before the **XIE_GET_*** event for that row.

You respond to the **XIE_GET_*** events by filling out the **data_rec** field of the **rec_request** structure.

For example, if your record handles are indices into an array of records stored in memory, you would probably return 0L in the **data_rec** field upon receiving the **XIE_GET_FIRST** event. In contrast, if your record handles are pointers to a linked list structure, you cast the record handle into a pointer, and copy the first database record you want in the list into that structure. In the third case, the **data_rec** field will already be filled in from the results of the **XIE_REC_ALLOCATE** event.

When your event handler receives an **XIE_GET_NEXT** event, XI is asking your application to fetch the data for the next record to be displayed in the list. The **XI_EVENT** structure in this event contains two fields that you care about: **spec_rec** and **data_rec**. The **spec_rec** field contains a record handle specifying a record from which you should get the next record. In the **data_rec** field of the structure, XI gives you a record handle. If your record handles represent indices in an array of records, you would set **data_rec** to '**spec_rec + 1**'. In contrast, if your record handles are pointers to records retrieved from disk, you cast the record handle into a pointer, and copy the next database record into that structure. Responding to **XIE_GET_FIRST** and **XIE_GET_NEXT** events is summarized in the following table.

XI Event	spec_rec (From XI)	data_rec (Set by Your Application)
XIE_GET_FIRST	N/A	Record 1
XIE_GET_NEXT	Record 1	Record 2
XIE_GET_NEXT	Record 2	Record 3
XIE_GET_NEXT	Record 3	Record 4

The reason why XI asks you to locate a record by giving you the record handle for a specific record and then asking you to retrieve the next record is that XI may not keep track of any records other than the records currently displayed on the list. Because record handles are abstract objects, and XI doesn't know how to manipulate them, XI *must* ask your application to find the next or previous record relative to a given record. This is exactly what the **XIE_GET_NEXT** event is requesting that your application do. It gives the event handler a record handle for a specified record and asks the application to get the record following it.

It is important to remember that **XIE_GET_FIRST** is not necessarily requesting the first record in the database. It can be the first record of any portion of the database that you care to display. For example, you may have a database of several hundred invoices for each of many vendors. Because you might want the list to contain the invoices for only one vendor, you would want to return a record handle to a copy of the

first invoice for the vendor when your application receives an **XIE_GET_FIRST** event, not necessarily the first invoice in the file. Likewise, you would want to refuse an **XIE_GET_NEXT** if you are already displaying the last record for that vendor.

Another point about the **XIE_GET_FIRST** event is that it may actually be asking for the a record part of the way through the list. If the list object has a vertical scroll bar and the user moves the thumb, the **XIE_GET_FIRST** event structure will contain a **percent** value that is not zero. In that case, you must return a record at that percentage. Otherwise, the thumb positioning will not work.

The application can refuse any of the **XIE_GET_*** events if no records should be read from the database. This could happen if you are at the end of the file, the database is empty, or you have read all of the records in the portion of the database to be displayed. If you do not want to display another record for these or other reasons, set the **refused** member of the event to **TRUE** indicating that another record will not be displayed.

Look at the event handler functions in the various example code modules for examples of these techniques.

8.4.4 Displaying Text

Whenever your event handler receives an **XIE_CELL_REQUEST** event, XI is asking you to give it text to display in a cell. You get **XIE_CELL_REQUEST** events after you've responded to **XIE_GET_*** events. For example, when initializing a list of two columns and two rows, you could get the following event stream:

```
XIE_GET_FIRST
XIE_CELL_REQUEST
XIE_CELL_REQUEST
XIE_GET_NEXT
XIE_CELL_REQUEST
XIE_CELL_REQUEST
```

Upon receiving an **XIE_CELL_REQUEST** event, your application will need to look in the **cell_request** structure of the **XI_EVENT**. In the **cell_request** structure, it will find out which column and record the request corresponds to and how long the string can be.

After locating the record and formatting the data into text, your application should then store that text in the **s** field of the **cell_request** structure. Be careful not to store more characters than the buffer can hold as indicated by the **len** field of the **cell_request** structure. **len** is the total buffer size, including the **'\0'**. Recall that **len** is sized when a list is defined, and can be changed after the list is instantiated by calling **xi_set_bufsize**.

XI will not generate **XIE_CELL_REQUEST** events for cells that are not visible if the preference **XI_PREF_OPTIMIZE_CELL_REQUESTS** is **TRUE**.

Refer to the section *Using Cells* for information (icons, color, fonts) on other options of the **XIE_CELL_REQUEST** event.

8.4.5 Processing User Input

As the user changes cell text and moves from cell to cell in the list, your event handler will receive many **XIE_CHAR_CELL**, **XIE_CHG_CELL**, and **XIE_OFF_CELL** events. These are the events that are important because they give you an opportunity to filter characters that the user is typing, verify the contents of cells, convert cell text to data and store the data in the copies of database records stored in memory.

You will also receive **XIE_ON_CELL** events, but these are rarely used, if ever.

8.4.5.1 Responding to XIE_CHAR_CELL and XIE_CHG_CELL Events

XIE_CHAR_CELL events are sent to your event handler when the user types characters into the cell. You can refuse these events to restrict the kinds of characters the user can type in a cell. You can also modify the character in this event to force a certain behavior such as converting all characters to uppercase. You'll find example code of character filtering in the *Using Cells* section of this chapter.

XIE_CHG_CELL events are sent whenever the user edits the contents of a cell. It doesn't tell you what changes were made or how the user made them. Instead, **XIE_CHG_CELL** only tells you that something happened to the cell. You cannot refuse **XIE_CHG_CELL** events because a change has already happened by the time you are notified of it.

8.4.5.2 Responding to XIE_OFF_CELL Events

When your event handler receives an **XIE_OFF_CELL** event, it is being notified that the user is done typing and wants to move the focus off of the cell. This gives you the opportunity to check that the user has entered valid data. For example, the user may be entering data for a database key field such as a part number or social security number. Before letting the focus move off of the cell, you will want to check that the part number or social security number is valid. In addition, it is a good idea to convert the cell text into data and store it in the appropriate record in memory.

It is important to remember you should wait to store the changes made to the contents of a cell until your event handler receives an **XIE_OFF_CELL** event. You cannot store changes in response to **XIE_CHG_CELL** events because partially entered data will not be valid. Instead, you should wait until your event handler receives an **XIE_OFF_CELL** event which notifies you that the user is attempting to move off of the cell.

For more information on how to validate cell data, and store record changes, see the code examples in the *Using Cells* section of this chapter. For code examples demonstrating how to read and write records to the database, see the *Interfacing to Databases* topic found later in this section.

8.4.6 Responding to Focus Movements

In addition to receiving **XIE_OFF_CELL** events, your event handler will receive other focus events as the user moves on and off of cells in a list.

XIE_OFF_COLUMN events are generated whenever the user presses the left or right arrow or tab key or scrolls the list with an up or down arrow. It would not be generated when the user pressed the up or down arrow within the currently displayed records of the list. This could be useful for applications that perhaps want to let the user enter a whole column of data, and then when they move off of the column to process it in some fashion.

XIE_OFF_ROW events are generated whenever the user presses the up or down arrow key, or presses the tab key in the last column.

Below are some examples of user actions and the resulting events.

1. Here are the events generated when the user presses the tab key, moving the focus from one cell to the next cell on the right:

```
XIE_OFF_CELL  
XIE_OFF_COLUMN  
XIE_ON_COLUMN  
XIE_ON_CELL
```

2. Here are the events generated when the user presses the down arrow key, moving the focus from one cell to the next cell below:

```
XIE_OFF_CELL
XIE_OFF_ROW
XIE_ON_ROW
XIE_ON_CELL
```

3. Here are the events generated when the user presses the down arrow key at the bottom of the list, causing the list to scroll:

```
XIE_OFF_CELL
XIE_OFF_COLUMN
XIE_OFF_ROW
XIE_OFF_LIST
XIE_REC_ALLOCATE
XIE_GET_NEXT
XIE_CELL_REQUEST (For each column)
XIE_ON_LIST
XIE_ON_ROW
XIE_ON_COLUMN
XIE_ON_CELL
```

8.4.7 Updating Databases When Using a List

Most of the time, an XI list will display fewer records than the database contains. Because of this, your application will need to keep a copy of the database records currently being displayed in memory. Also, it will need to write the ones changed by the user out to the database. Keep in mind that XI will tell your application when to read records, but not when to write them. Because of this, there are two issues you must address when interfacing to a database. You must figure out when to update the memory copy of your records and when to store updated records in the database. In the following discussion, we describe the opportunities XI gives you to write records and check that the user has entered valid data. Keep in mind that our suggestions for interfacing to the database are only suggestions and may not be appropriate for your application.

8.4.7.1 Getting the Record Handle

As you saw above, you get record handles from **XIE_GET_*** and **XIE_CELL_REQUEST** events. However, before you can respond to the opportunities XI gives you to update records and write them to the database, you will need to know how to find the record that corresponds to the row of interest upon receiving an event that does not contain a record handle in its **XI_EVENT** structure.

In the method of interfacing to the database described here, the events that you will need to respond to that don't have record handles sent with them are **XIE_CHAR_CELL**, **XIE_CHG_CELL**, **XIE_OFF_CELL** or **XIE_OFF_ROW** events. If your event handler receives one of these events, your application will need to look in the **XI_OBJ** structure sent with the event to find the row number. Once you know the row number, you can get the corresponding record handle by calling `xi_get_list_info`**xi_get_list_info**. **xi_get_list_info** returns an array of record handles. By indexing into the array with the row number, you can find the record handle associated with the row of interest.

The following function, from "lstdb.c", demonstrates this process.

```
static long row_to_record( XI_OBJ* list, int row_num )
{
    int count;

    long* handles = xi_get_list_info( list, &count );
    return handles[ row_num ];
}
```

You will see examples of calling this function in the code below.

8.4.7.2 Marking a Record as Edited

As the user makes changes to text, you will need to track those changes so that you can make sure that the changes are valid and write modified records to the database at appropriate times. We recommend you do this by allocating two extra **BOOLEAN** flags associated with the application data for the interface. These flags will be used to track the status of each record as the user makes changes.

The information these flags will contain is the status of the text displayed on the list relative to the record in memory, and the status of the record in memory relative to the record in the database. In one flag, you keep track of the “edited” state so that you can know when the text of the cell with the focus is different than the record stored in memory. Likewise, in the second flag, you will need to keep track of the “modified” state so that you can know when the record in memory for the current row is different than the one in the database. In the example code, these flags are called **cell_changed** and **row_changed**.

When your application receives an **XIE_CHG_CELL** event for a cell, it will need to set the edited flag to **TRUE** to mark the record as “edited”. The following code, from “lstdb.c”, demonstrates this.

```
case XIE_CHG_CELL:
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

    list_info->cell_changed = TRUE;
    break;
}
```

8.4.7.3 Validating Cell Text

As explained earlier in this section, when your event handler receives an **XIE_OFF_CELL** event, your application is being notified that the user is finished typing text in a cell and wants to move the focus to another place on the interface. At this point your application should verify that text entered is valid and update the copy of the record stored in memory. If the cell contains valid data, the record will need to be marked as modified by setting the modified flag to **TRUE**. At this time, the edited flag should be reset to **FALSE** to indicate that the text displayed in the row now represents the record data stored in memory.

If the text does not pass validation, your application should refuse the **XIE_OFF_CELL** event. After refusing the event you could either reset the text in the cell to what it was before the user made changes, or show him that he must try again by selecting the text in the cell. To reset the text in the cell, use **xi_set_textxi_set_text**. To select the cell text, call **xi_set_selxi_set_sel**.

If the text was invalid and you chose to reset the text, you will need to mark the record as not edited by resetting the edited flag to **FALSE**. If you simply select the cell, you will need to keep the record marked as edited. This tells you that text currently being displayed still represents different data than the record kept in memory.

The following code, from “lstdb.c”, demonstrates processing the **XIE_OFF_CELL** event.

```

case XIE_OFF_CELL:
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

    if ( list_info->cell_changed )
    {
        XI_OBJ* cell = xiev->v.xi_obj;

        if ( !emp_set_value( row_to_record( cell->parent,
            cell->v.cell.row ), column_to_field( cell->parent,
            cell->v.cell.column ), xi_get_text( cell, NULL, 0 ) ) )
        {
            xiev->refused = TRUE;
            xi_set_sel( cell, 0, INT_MAX );
        }
        else
        {
            list_info->cell_changed = FALSE;
            list_info->row_changed = TRUE;
            xi_cell_request( cell );
        }
    }
    break;
}
}

```

8.4.7.4 Marking a Record as Modified

The second flag indicates that the record stored in memory has been modified and no longer matches the record in the database. You should set this flag to **TRUE** only after you have verified the text upon receiving an **XIE_OFF_CELL** for a record marked as edited, and have updated the copy of the record kept in memory. Since the text displayed on the screen now reflects the data stored in memory, you will also need to reset the edited flag to **FALSE** indicating its new state. This is shown in the example code above.

8.4.7.5 Writing Database Records

The best time for an application to write a record marked as modified to the database is in response to an **XIE_OFF_ROW** event. After writing the record, it should reset the modified flag to **FALSE** to indicate that the memory copy of the record contains the same data as the one in the database.

The following code, from “lstdb.c”, demonstrates this.

```

case XIE_OFF_ROW:
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

    if ( list_info->row_changed )
    {
        XI_OBJ* row = xiev->v.xi_obj;

        emp_update( row_to_record( row->parent, row->v.row_data.row ) );
        list_info->row_changed = FALSE;
    }
    break;
}
}

```

8.4.8 Scrolling the List

When users want to view records that are not currently displayed on the list, they will attempt to scroll the list. For example, suppose that a user is entering records in a database application. The user could attempt to scroll the list by trying to move the focus off the bottom or top of the list by pressing a tab, backtab or an

arrow key. In addition, the user could try to scroll the list by clicking on a button or by using a scroll bar if it is provided by the application developer. Regardless of the method the user uses, XI will generate the same events that are typically seen in the normal operation of the list. It will generate focus events to check that it is OK to move the focus, and record request events requesting that the application read records from the database and fill the corresponding rows with text.

If the user is attempting to scroll the list by pressing either the tab, backtab or arrow keys, XI will automatically scroll the list. However, if the user is attempting to scroll by using a button or scroll bar provided by you, then you will need to scroll the list manually by calling `xi_scrollxi_scroll`.

In the following discussion, you will learn how to respond to the events generated when XI scrolls the list automatically, how to call `xi_scroll` yourself, and what sequence of events to expect when you scroll the list yourself. You will find out what happens to the focus as the list scrolls. In addition, you'll find some tips on verifying records before letting the user scroll the list, and will see some example code demonstrating how to handle the various things that can happen when the user wants to scroll a list.

8.4.8.1 Events Sent While Scrolling

To illustrate the kinds of events your application can receive while scrolling, let's suppose that the user tries to scroll the list by pressing the down arrow key. To notify you of this, the first event XI sends you is an **XIE_OFF_CELL**. This event indicates that the user has finished typing in a cell and gives your application the opportunity to check that the text he entered is valid, and to store the valid text as data in a record kept in memory. Following the **XIE_OFF_CELL** event, the next event your event handler will receive is an **XIE_OFF_ROW** event. This event gives you the opportunity to write the copy of the record stored in memory to the database if your application needs to. After receiving the **XIE_OFF_ROW** event, your event handler may receive **XIE_OFF_COLUMN** and **XIE_OFF_LIST** events. These are generated because XI will move the focus off of the list before automatically scrolling the list. You do not need to respond to these.

After XI is finished generating focus events, your event handler will receive an **XIE_REC_ALLOCATE** event asking your application to allocate space for a record. The **XIE_REC_ALLOCATE** event is followed by the **XIE_GET_NEXTXIE_GET_NEXT** event asking your application to read a record. The **XIE_GET_NEXT** event is followed by **XIE_CELL_REQUESTXIE_CELL_REQUEST** events requesting text for each visible cell in the row. After the records have been requested and cells have been filled with text, XI will then generate **XIE_ON_*** events to put the focus in the correct location on the list. A summary of the events generated when XI scrolls is shown below.

```
XIE_OFF_CELL
XIE_OFF_COLUMN
XIE_OFF_ROW
XIE_OFF_LIST
XIE_REC_ALLOCATE
XIE_GET_NEXT
XIE_CELL_REQUEST (For each column)
XIE_ON_LIST
XIE_ON_ROW
XIE_ON_COLUMN
XIE_ON_CELL
```

Suppose that in the previous example the user attempted to scroll off of the top of the list instead of the bottom. In this case, the only difference in the type of events your event handler would receive is that it would get an **XIE_GET_PREV** event instead of an **XIE_GET_NEXT** event. An **XIE_GET_PREV** event is identical to an **XIE_GET_NEXT** event except that it asks your application to provide the previous record instead of the next one.

8.4.8.2 Scrolling Under Program Control

To scroll the list from your application, you will need to call `xi_scroll` and pass it the list object and either a positive or negative number of rows to scroll. In turn, XI will generate `XI_GET_*` events asking you to get that number of records.

An important thing to remember is that `xi_scroll` may not scroll the number of lines that you request. For example, if you asked it to scroll up five lines, it will ask for the five records previous to the first one currently displayed in the list. If you are at the beginning of the file and cannot read any more records before the one you just read, you can refuse one of the `XIE_GET_PREV` events. When this happens, XI won't try to fill up those extra rows with blank space. Instead, it will simply scroll as many rows as were available while requesting records for them. When it is done, it will return the number of rows that were actually scrolled. Therefore, if you asked it to scroll up five rows, `xi_scroll` might return 3, indicating that it could only scroll up three lines. When XI is done requesting records, XI will redisplay the list by using the XVT function `xvt_dwin_scroll_rect`. It then issues `XI_CELL_REQUEST` events asking for text to display in the newly exposed rows.

8.4.8.3 Scrolling the List to a Specific Record

To scroll the list to a specific record, there is a function that is similar to `xi_scroll`. This function is `xi_scroll_rec`. The difference between `xi_scroll` and `xi_scroll_rec` is that `xi_scroll_rec` takes a record handle as an argument. That record handle will be the first row shown on the list.

`xi_scroll_rec` will cause `XIE_GET_NEXT` events to get the records after the one supplied by the function. `XIE_CELL_REQUEST` events will occur for the handle supplied by `xi_scroll_rec` as well as the other records that were requested.

8.4.8.4 Placing a Vertical Scroll Bar on a List

To put a vertical scroll bar on a list, after calling the convenience function `xi_add_list_def`, you should set the `scroll_bar` field in the object definition to `TRUE`. It is very common to use a vertical scroll bar on a list, but it requires some extra work to support it.

You will need to respond to the `XIE_GET_FIRST` event slightly differently. When a list has a scroll bar, the user may move the scroll bar thumb to an arbitrary position in the scroll bar. When they do this, your event handler will receive an `XIE_GET_FIRST` event. There is a `percent` field in the `rec_request` structure. This field indicates the percentage through your data for the record to be placed in the `data_rec` field of the `rec_request` member. (For an `XIE_GET_FIRST` that was not generated as a result of moving the thumb, the `percent` field will always be zero.)

There is a special case. If the thumb was dropped at the bottom of the scroll bar, then XI generates an `XIE_GET_LAST` event, followed by `XIE_GET_PREV` events.

The following function, from “datdb.c”, is called in response to an `XIE_GET_FIRST` event. It demonstrates how to get the proper record when the `percent` field may be set.

```

BOOLEAN emp_get_first( XI_EVENT* xiev )
{
    EMPREC* rec = (EMPREC*)xiev->v.rec_request.data_rec;

    if ( emp_file == NULL )
        return FALSE;
    if ( xiev->v.rec_request.percent == 0 )
    {
        rec->rec_num = db_first( emp_file, rec );
        rec->position = 0;
    }
    else
    {
        long rec_count = db_get_record_count( emp_file );

        if ( rec_count == 0 )
            return FALSE;
        rec->position = rec_count * xiev->v.rec_request.percent / 100;
        rec->rec_num = db_goto( emp_file, rec->position, rec );
    }
    if ( rec->rec_num == NULL_REC_NUM )
        return FALSE;
    return TRUE;
}

```

In addition, there is another event to which you will need to respond. When the user presses the up arrow key and causes the list to scroll, the thumb on the scroll bar should move to reflect the percentage through the database. To implement this behavior, XI sends an event, **XIE_GET_PERCENT**, to request the percentage through the database of a specific record. The pertinent record is in the **XI_EVENT** structure. XI actually sends two of these events, one for the first visible record, and one for the last visible record. From this, XI can calculate the position of the thumb. In addition, XI can calculate the thumb proportion on systems that support this.

The following function, from “datdb.c”, is called in response to an **XIE_GET_PERCENT** event. It demonstrates how to respond properly to that event.

```

void emp_get_percent( XI_EVENT* xiev )
{
    EMPREC* rec = (EMPREC*)xiev->v.get_percent.record;
    long rec_count = db_get_record_count( emp_file );

    if ( rec_count > 1 )
    {
        xiev->v.get_percent.percent = (int)( rec->position * 100
                                             / ( rec_count - 1 ) );
        if ( xiev->v.get_percent.percent == 0 && rec->position != 0 )
            xiev->v.get_percent.percent = 1;
    }
}

```

8.4.9 Changing List Attributes

You can dynamically change the look and feel of a list by changing its attributes and the attributes of the columns, rows and cells contained within it. In XI, attributes affect certain aspects of an object’s behavior and how it is displayed. For a summary of the attributes a list can have look in *Characteristics of XI Objects*.

To change the attributes of an XI object, you call **xi_set_attrib** with a pointer to the object whose attributes you want to change, and a bitwise OR’ed combination of values corresponding to the new attributes you want the object to have. Like rows and columns, you cannot change the attributes of a list while a cell in the list has the focus. To move the focus to a safe place, call **xi_move_focus** passing to it the interface object. Nothing special happens when the interface gains the focus, it is just a convenient place to

park it while you change the attributes of an XI object. When `xi_set_attrib` returns, you can move the focus back to the list if you haven't disabled it or made it invisible.

When you want to change the look and feel of the entire list, the attributes you can set for a list are: `XI_ATR_ENABLED`, `XI_ATR_VISIBLE`, `XI_ATR_TABWRAP`, and `XI_ATR_NAVIGATE`. If you want to know what attributes a list has, you call `xi_get_attrib`. It returns the current attribute bit values OR'ed together. If you want to change the look and feel of individual columns and rows in the list, read the *Using Columns* and *Using Rows* sections found later in this chapter.

8.4.9.1 Changing a Single Attribute

In most cases, you will want to change only one attribute of the list such as disabling the list or making it invisible. You would call `xi_get_attrib` to get the attributes for the list, change the bit and then call `xi_set_attrib` to give the list its new look. The following code segment demonstrates setting the enabled attribute:

```
xi_set_attrib(xi_obj, xi_get_attrib(list_obj) | XI_ATR_ENABLED);
```

The following code segment demonstrates disabling a list by clearing the enabled attribute:

```
xi_set_attrib(xi_obj, xi_get_attrib(list_obj) & ~XI_ATR_ENABLED);
```

8.4.10 Changing the List Size

Sometimes, when the user resizes the window, you may want the list contained in the window to resize with the window.

You can enable this behavior by setting `list_def->v.list->resize_with_window` to `TRUE` after calling `xi_add_list_def`. In this case, XI will resize the list so that the lower right corner of the list is always placed in the lower right corner of the client area of the window.

However, you may decide that you wish to only size the horizontal component or the vertical component of the list when the window resizes. To implement this behavior, you can call the function `xi_set_list_size` upon the `E_SIZE` event.

The following code, from "ltsync.c", demonstrates manual sizing of lists using the `xi_set_list_size` function.

```
case XIE_XVT_EVENT:
    if ( xiev->v.xvte.type == E_SIZE )
    {
        XI_OBJ* list;
        RCT      rct;

        list = xi_get_obj( itf, LIST2_CID );
        xi_get_rect( list, &rct );
        if ( rct.right != xiev->v.xvte.v.size.width )
            xi_set_list_size( list, rct.bottom - rct.top,
                             xiev->v.xvte.v.size.width - rct.left );

        list = xi_get_obj( itf, LIST3_CID );
        xi_get_rect( list, &rct );
        if ( rct.bottom != xiev->v.xvte.v.size.height )
            xi_set_list_size( list, xiev->v.xvte.v.size.height - rct.top,
                             rct.right - rct.left );
    }
}
```

```

list = xi_get_obj( itf, LIST4_CID );
xi_get_rect( list, &rct );
if ( rct.bottom != xiev->v.xvte.v.size.height
    || rct.right != xiev->v.xvte.v.size.width )
    xi_set_list_size( list, xiev->v.xvte.v.size.height - rct.top,
                    xiev->v.xvte.v.size.width - rct.left );
}
break;

```

8.5 Using Cells

Unlike most other XI objects, cells are never instantiated. Instead, they are created implicitly whenever you instantiate a list. For example, if a list has five columns and ten rows, it will have fifty cells. Although it might be useful to instantiate cells, the reason XI does not create real cell objects is to improve the performance of your application and to save memory.

Because cells are not real objects, they have to be “manufactured” either by your application or by XI. There are five times XI will make a cell “pseudo-object” for you:

1. When it notifies you of typing in the cell by sending an **XIE_CHAR_CELL** event.
2. When it notifies you that the contents of a cell has changed by sending an **XIE_CHG_CELL** event.
3. When it notifies you that a cell button has been pressed by sending an **XIE_BUTTON** event.
4. When it sends either an **XIE_ON_CELL** or **XIE_OFF_CELL** to notify you that the user wants to move the focus on or off a cell.
5. When you call **xi_get_focus** and the focus is currently in a cell.

When sending you any of these events, XI manufactures a cell object and gives you a pointer to this object. You can use this pointer to call XI functions that take an XI object such as **xi_set_text**, **xi_get_text** and **xi_get_sel**. It is important to note that you cannot use this cell object for long. Any of the above cases could cause XI to reuse the memory to make another cell object.

Whenever your application needs to explicitly refer to a cell, but XI hasn’t made a cell object for you to use, you will need to manufacture one yourself. For example, suppose that your application wants to set the text of a cell during the off focus event for a different cell. (You may want to do this because the value is computed.) In order to call, **xi_set_text**, you must have a cell object. You will need to use the **XI_MAKE_CELL** macro to make a temporary cell object that will be passed to the function.

8.5.1 Cell Request Events

As we mentioned before, after XI generates record request events (**XIE_GET_***), it generates cell request events (**XIE_CELL_REQUEST**). Cell request events ask the application for all the information necessary to display a cell. The information you can supply to XI about a cell is:

- The text for the cell. The source of the text for the cell will be in your own format. You will need to convert your information to text.
- The icon for the cell if the cell is going to contain an icon instead of text. You can either have text or an icon in a cell, but not both.
- The attribute of the cell. You can specify that the cell is selected.
- The foreground color of the cell.
- The background color of the cell.
- The font of the cell.

- Whether the cell has a cell button, if the cell button is on the right or on the left, and if the cell button is only visible if the cell has the focus.

Refer to the *XI Programmer's Reference* for complete details on responding to an **XIE_CELL_REQUEST** event. Refer to the example program, especially the “Memory” list, for samples of how to respond to the **XIE_CELL_REQUEST** event.

In some situations, you may have changed your data structures underlying the list, and you might wish to have the list reflect your changes. You can force XI to generate **XIE_CELL_REQUEST** events by calling the function **xi_cell_request**. You can pass four types of objects to **xi_cell_request**: **XIT_LIST**, **XIT_COLUMN**, **XIT_ROW**, and **XIT_CELL**. If you pass an object of type **XIT_LIST**, then XI will generate **XIE_CELL_REQUEST** events for every cell in the list. If you pass an object of type **XIT_COLUMN**, then XI will generate **XIE_CELL_REQUEST** events for every cell in the column. If you pass an object of type **XIT_ROW**, then XI will generate **XIE_CELL_REQUEST** events for every cell in the row. If you pass an object of type **XIT_CELL**, then XI will generate exactly one **XIE_CELL_REQUEST** event for the cell that you specified.

8.5.2 Making Cell Objects

When you use a cell, you will have to have a way of referencing it. Often you can get a pointer for a cell object when receiving a cell event as discussed above. However, there are times when you will need to make one yourself. This is done by using the **XI_MAKE_CELL** macro to manufacture a “cell object”.

Whenever you use the **XI_MAKE_CELL** macro to manufacture a cell object, you pass the address of an **XI_OBJ** structure to it. XI uses the structure to make a valid cell object. Once you’ve got one of these cell pseudo-objects, you can pass its pointer to any XI function that takes a cell object.

In the following code, from “lstmem.c”, we respond to a row selection by changing the icon for a cell. We then need to manufacture a cell object in order to call **xi_cell_request** to update the display.

```
case XIE_SELECT:
    switch ( xiev->v.select.xi_obj->type )
    {
        case XIT_ROW:
        {
            XI_OBJ* list = xiev->v.select.xi_obj->parent;
            int row = xiev->v.select.xi_obj->v.row_data.row;

            if ( column_to_code( list, xiev->v.select.column )
                == VALUE_IN_STOCK )
            {
                XI_OBJ cell;

                mem_change_stock( row_to_record( list, row ) );
                XI_MAKE_CELL( &cell, list, row, xiev->v.select.column );
                xiev->refused = TRUE;
                xi_cell_request( &cell );
            } else
                mem_select_row( row_to_record( list, row ),
                               xiev->v.select.selected );
        }
        break;
    }
}
```

In general, you should not keep a cell object around. As the list scrolls, the row and column for the cell object change, and are probably not what you intended when you used the **XI_MAKE_CELL** macro. The best practice is to use an automatic variable for the object.

8.5.3 Being Notified of Typing in a Cell

When the user types into a cell on a list, the event handler containing the list will receive an **XIE_CHAR_CELL** event. Your application can refuse this event to disallow the insertion of the character into the cell, or modify the character to force certain kinds of behavior. This event is most often used to restrict the kinds of characters the user can type into a cell, such as only letters for a name, digits for a part number, digits separated by '/' for dates, etc.

When the user changes the contents of a cell, the event handler containing the list will receive an **XIE_CHG_CELL** event. Since the purpose of this event is to indicate that the contents of the cell has changed, you will not be told what keystrokes or commands were used to make the change. Instead, **XIE_CHG_CELL** only tells you that something happened to the cell.

8.5.3.1 Character filtering

To limit the kinds of characters the user can type, you need to respond to the **XIE_CHAR_CELL** event. To display only the allowable characters in the cell, you examine the **xiev->v.chr.ch** field and refuse the event for any invalid characters.

In the following example, from "lstlink.c", we demonstrate how this is done.

```
case XIE_CHAR_CELL:
{
    XI_OBJ*    cell = xiev->v.chr.xi_obj;
    LINK_FIELD field = column_to_field( cell->parent,
                                       cell->v.cell.column );

    if ( field == LINK_DATE )
        xiev->refused = !validate_date_char( xiev->v.chr.ch );
    break;
}
```

8.5.4 Validating Cell Text

In addition to receiving an **XIE_CHAR_CELL** event, another time you'd want to look at the contents of a cell is to check that something reasonable was entered before allowing the user to move off of the cell. For example, the user may be entering data for a database key field such as a part number or social security number. In this case, you will want to check the database to verify that the user entered a valid database key before letting the user continue. You could also check for valid dates, currency amounts and other kinds of data.

The basic idea here is that you let the user freely edit the contents of a cell until he tries to move the focus to another object in the interface. Users can move the focus by either tabbing off of the cell, or clicking the mouse on another cell or edit field. When the user tries to change the focus to a different object in the window, your event handler will be notified of this by an **XIE_OFF_CELL** event.

It is important to note that you do not receive an **XIE_OFF_CELL** event when the user switches to another window. This is because the user may want to switch to another window to browse through some other data. XI makes sure that a focus event doesn't occur, and the interface is in the same state as before the user left.

8.5.4.1 Checking for Valid Data

As mentioned above, when the user attempts to move off of the cell, your event handler for the interface receives an **XIE_OFF_CELL** event. In response, you should call **xi_get_text** on the cell, and hand the text it returns to a validation routine.

If your validation routine rejects the data, then you'll want to refuse the **XIE_OFF_CELL** event by setting the refused member of the **XI_EVENT** structure to **TRUE**. This will tell XI not to move the focus after all. To inform the user that the text is invalid, you might want to beep by calling the XVT function **xvt_beep**, and select the text by calling **xi_set_sel**. This will result in highlighting the text so that if the user starts typing, they overwrite the previous text.

The following code, from "lstdb.c", demonstrates this.

```
case XIE_OFF_CELL:
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

    if ( list_info->cell_changed )
    {
        XI_OBJ* cell = xiev->v.xi_obj;

        if ( !emp_set_value( row_to_record( cell->parent,
            cell->v.cell.row ), column_to_field( cell->parent,
            cell->v.cell.column ), xi_get_text( cell, NULL, 0 ) ) )
        {
            xiev->refused = TRUE;
            xi_set_sel( cell, 0, INT_MAX );
        } else
        {
            list_info->cell_changed = FALSE;
            list_info->row_changed = TRUE;
            xi_cell_request( cell );
        }
    }
}
break;
}
```

If the text entered by the user passes validation, then your application will want to convert the text to data and store it in the memory copy of the record. To do this, the application will need to call **xi_get_list_info** to get the array of record handles for the list, and index into that array with the row number found in the **xi_obj->v.cell.row** field of the **XI_EVENT** structure. This will give your application the record handle for the row of interest, and it can use that handle to store the converted data. The previous code example calls this function in the **row_to_record** function, as follows.

```
static long row_to_record( XI_OBJ* list, int row_num )
{
    int count;

    long* handles = xi_get_list_info( list, &count );
    return handles[ row_num ];
}
```

It is important to remember that when your event handler receives an **XIE_CHG_CELL** event, it is not a good time to convert the text into data and store it in the data record, because partially entered data will not be valid. Instead, you should wait until your event handler receives an **XIE_OFF_CELL** event notifying your application that the user is attempting to move off of the cell. This is the appropriate time to validate the data and store it in its record.

8.6 Using Rows

Like cells, rows are never instantiated. Instead, they are created implicitly whenever you instantiate a list. Although it might be useful to instantiate rows, the reason XI does not create real row objects is to improve the performance of your application and to save memory.

Because rows are not real objects they have to be "manufactured" either by your application or by XI. XI will make a row for you when it sends your event handler **XIE_ON_ROW**, **XIE_OFF_ROW**, or **XIE_SELECT** events. notifying your application that the user wants to move the focus on or off a row.

When sending you these events, XI manufactures a row object and gives your event handler a pointer to this object in the **xi_obj** field in the **XI_EVENT** structure. You can use this pointer to call XI functions that take a row object such as **xi_delete_row**, **xi_set_attrib** and **xi_get_attrib**. It is important to note that you cannot use the row object once the event handler has returned because XI will reuse the memory to make another row object.

8.6.1 Responding to Record Request Events

When your application responds to record request events, it has the option of supplying more information to XI than just the record handle. On a record request event you can supply:

- The record handle.
- The attribute of the row. You can specify that the row is selected.
- The color of the row.
- The height of the row.

Refer to the *XI Programmer's Reference* for complete details on responding to **XIE_GET_*** events.

8.6.2 Making Row Objects

There are two times that you would want to manufacture a row object. You would use the row object to set the attributes for a row when calling **xi_set_attrib** and **xi_get_attrib**, and when deleting the contents of the row by calling **xi_delete_row**. In this section, you will find information and code examples for manufacturing row objects and responding to focus events. Specifically, you will see examples of how to delete the contents of a row, validate the contents of a row, and set its attributes.

When you use a row, you will have to have a way of referencing it. Often you can get a pointer for a row object when receiving a row event as discussed above. However, there are times when you will need to make one yourself. This is done by using the **XI_MAKE_ROW** macro to manufacture a “row object”.

Whenever you use the **XI_MAKE_ROW** macro to manufacture a row object, you pass the address of an **XI_OBJ** structure to it. XI uses the structure to make a valid row object. Once you've got one of these row pseudo-objects, you can pass its identifier to any XI function that takes a row object.

The following code, from “lstlink.c”, demonstrates using **XI_MAKE_ROW** to manufacture row objects in order to get their attribute.

```
static long get_unselected_handle( XI_OBJ* list )
{
    int      count, num;
    long*    handles;

    handles = xi_get_list_info( list, &count );
    for ( num = 0; num < count; num++ )
    {
        XI_OBJ row;

        XI_MAKE_ROW( &row, list, num );
        if ( !( xi_get_attrib( &row ) & XI_ATR_SELECTED ) )
            return handles[ num ];
    }
    return 0;
}
```

In general, you should not keep a row object around. As the list scrolls, the row changes, and is probably not what you intended when you used the **XI_MAKE_ROW** macro. The best practice is to use an automatic variable for the object.

8.6.3 Deleting a Row

There are two ways to think about deleting a row. One way is to delete the contents of a row, and the other is to physically make the list shorter by one row. In XI, when we say you can delete a row, we mean that you can remove the contents of a row from the list, not resize the list. Since XI lists scroll in the vertical direction, the final result of deleting the contents of a row is to cause the list to scroll to fill in the empty space while keeping the displayed size of the list fixed.

The function you use to delete the contents of a row is `xi_delete_row`. Calling `xi_delete_row` causes XI to remove the row from the list and scroll the rows below it up one row to fill in the empty space. As a result of the scrolling operation, XI will send your event handler an `XIE_GET_NEXT` event asking it to get a record to display in the bottom row of the list. Following the `XIE_GET_NEXT` event, it asks you to fill the cells with text by sending you `XIE_CELL_REQUEST` events. Thus the final result of calling `xi_delete_row` is removing a row from the list and filling the space created by scrolling the list.

The following code, from “`lstlink.c`”, demonstrates deleting a row.

```
case DELETE_CUR_CID:
{
    XI_OBJ* obj = xi_get_focus( itf );

    if ( obj->type == XIT_CELL )
    {
        XI_OBJ row;

        link_delete( row_to_record( obj->parent, obj->v.cell.row ) );
        XI_MAKE_ROW( &row, obj->parent, obj->v.cell.row );
        xi_delete_row( &row );
        update_numbers( obj->parent );
    }
    break;
}
```

8.6.4 Inserting a Row

The function you use to insert a row is `xi_insert_row`. Calling `xi_insert_row` causes XI to insert space for a row in the list, then generate a record request to retrieve a record for the inserted row.

If the record is the first record in the list that XI knows about, XI generates an `XIE_GET_PREV` event to retrieve the record. If the record is other than the first record, XI generates an `XIE_GET_NEXT` event to retrieve the record. If the list is empty, and has no records, then XI generates an `XIE_GET_FIRST` event.

The following code, from “`lstlink.c`”, demonstrates inserting a row.

```

case ADD_ONE_CID:
{
  XI_OBJ* obj = xi_get_focus( itf );

  if ( obj->type == XIT_CELL )
  {
    add_records( list_info->link_list, row_to_record( obj->parent,
      obj->v.cell.row ), 1 );
    xi_insert_row( obj->parent, obj->v.cell.row + 1 );
  }
  else
  {
    add_records( list_info->link_list, 0L, 1 );
    refresh_list( xi_get_obj( itf, LIST_CID ) );
  }
  update_numbers( xi_get_obj( itf, LIST_CID ) );
  break;
}

```

8.6.5 Validating the Contents of a Row

Sometimes an application will want to validate the data for a row as a whole before allowing the user to move the focus to another object on the interface. It is important to remember that when your event handler receives **XIE_OFF_CELL** events, it is not a good time to save the memory copy of the record to the database. This is because you will be updating the memory copy of the record each time the user tries to move off of a cell. Instead, you should wait until you receive an **XIE_OFF_ROW** event notifying you that the user is attempting to move off of the row and is done editing the record.

8.6.5.1 Checking for Valid Data

When the user attempts to move off of the row, your event handler will receive an **XIE_OFF_ROW** event. In response, you will need to find out the row number by looking in the **XI_EVENT** structure (**xiev->xi_obj->v.row**). Once you know the row number you can manufacture cell objects for any cell in the row by using **XI_MAKE_CELL** macro. Making cell objects are discussed in detail in the *Using Cells* section of this chapter. When you have cell objects you can call **xi_get_text** to get the text for those cells in the row, and perform your validation routine.

Note that cells are usually validated in response to the **XIE_OFF_CELL** event, but you may need to validate the row as a whole.

If the text in the cells of the row is invalid, then you should refuse the **XIE_OFF_ROW** event. Depending on what is appropriate for your application, you then notify the user that there is a problem by either resetting the text for the entire row or indicating that the user must re-enter the text for a specific cell. You might also want to beep by calling the XVT function **xvt_beep**.

If you chose to set the text for each cell in a row, you will need to reread the record from the database, manufacture a cell object for each cell in the row, and reset the contents of the cells by calling **xi_set_text**. In contrast, if you chose to indicate that the user needs to re-enter the text in a specific cell, you will need to move the focus to the cell and select it by calling **xi_set_sel**.

8.7 Using Columns

The most common use of column objects is to change the look and feel of a list. Although there are some attributes you can apply to affect the whole list, the character of a list is really determined by the nature of the columns that it contains.

In this section, we will illustrate how to change the way a list looks by changing column headings, changing column widths, adding and deleting columns and by giving the columns different attributes. Code examples will be provided to give you a better idea of the options you have when using column objects.

8.7.1 Getting a Column Object

To change the width and heading of a column, you will need a pointer to the column object. If you have a control ID for the column, you can call **xi_get_obj** with the control ID to get the pointer to the column object. If you have a pointer to the list containing the column, you can call **xi_get_member_list** to get an array of pointers to the columns in the list. Once you have the array, you can search through it to locate the pointer for the column. If you have a cell object, you can get the list object, call **xi_get_member_list** to get an array of pointers to the columns in the list, and index into the list with the column number stored in the cell object.

8.7.2 Changing a Column's Heading

To change a column's heading, you will need to call **xi_set_text** with a pointer to the column object you're interested in. If you are changing the text in response to an **XIE_OFF_COLUMN** event, your event handler can use the pointer sent with the event. Otherwise, you can get the pointer to a specific column by either using control ID for the column, or pointer to the list as explained above.

8.7.3 Changing the Width of a Column

Your application may allow users to change the width of a column to display different kinds of data in a column. If this is the case, you can either change the width of the column to accommodate longer strings of text, or allow cells in the column to contain more text than can be displayed by setting the attribute, **XI_ATR_AUTOSCROLL** for the column. If your chosen look and feel is changing the column width, keep in mind that you cannot change the width of a column when a cell in the column has the focus. If a cell has the focus in the column of interest, you will need to move the focus to the interface by calling **xi_move_focus** with the interface object. Moving the focus to the interface generates the appropriate **XIE_OFF_*** events, but no **XIE_ON_*** events.

To reset the width of a column, you will need to get the pointer to the column object as explained above. Once you have the object pointer, you will need to call **xi_set_column_width** passing it the pointer and an integer width in form units. Changing the width of a column is demonstrated in the following code example:

```
column_obj = xi_get_obj( itf, COLUMN1_CID );
xi_set_column_width( column_obj, width_in_chars * XI_FU_MULTIPLE );
```

8.7.4 Changing Column Attributes

Column attributes determine the look and feel of all cells in a column. Their look and feel may also be effected by whether or not the row they are a part of is enabled or disabled. You can dynamically change column attributes by calling **xi_set_attrib** with a pointer to the column object and a bitwise OR'ed combination of values corresponding to the new attributes you want the column to have.

As when deleting a column or changing its width, you cannot change the attributes of a column while a cell in the column has the focus. We recommend that you move the focus to the interface while setting the attributes for the column, and move it back to the cell after **xi_set_attrib** returns. To move the focus to the interface, call **xi_move_focus** with the interface object. To move the focus back to the cell, call **xi_move_focus** with a cell object. Recall that you can get a cell object from an XI event or can manufacture one yourself using the **XI_MAKE_CELL** macro. Manufacturing cell objects is explained in *Using Cells*.

The attributes you can set for the cells in a column are : **XI_ATR_ENABLED**, **XI_ATR_EDITMENU**, **XI_ATR_AUTOSCROLL**, **XI_ATR_AUTOSELECT**, **XI_ATR_RJUST**, **XI_ATR_PASSWORD**, **XI_ATR_SELECTED** and **XI_ATR_READONLY**. If you want to know what attributes a particular column has, you call **xi_get_attrib**. It returns the current attribute bit values OR'ed together. See

Characteristics of XI Objects for a description of how each attribute affects the behavior of cells in a column.

8.7.4.1 Changing a Single Attribute

In most cases, you will want to change only one attribute of the column such as disabling the column or making it “read only”. To apply the attribute, you would call **xi_get_attrib** to get the current attributes for the column, change the relevant bit and then call **xi_set_attrib** to give the column its new look. The following example illustrates disabling a column.

```
obj = xi_get_obj(itf, COLUMN1_CID);
xi_set_attrib(obj, xi_get_attrib(obj) & ~XI_ATR_ENABLED);
```

The following example illustrates enabling a column.

```
obj = xi_get_obj(itf, COLUMN1_CID);
xi_set_attrib(obj, xi_get_attrib(obj) | XI_ATR_ENABLED);
```

8.7.5 Column Events

Users can dynamically adjust columns in a list. They can delete columns by dragging and dropping the column heading off of the list. They can move columns around by dragging and dropping the column heading. They can resize columns by grabbing and dragging the border between the column headings.

Each one of these operations generates an event. The events are **XIE_COL_DELETE**, **XIE_COL_MOVE**, and **XIE_COL_SIZE**. Information about which column will be affected, and how it will be affected is supplied with these events. In addition, these events are refusable, allowing you to customize your application and specify which columns can be deleted, moved, or sized. See the *XI Programmer's Reference* for more details on these events.

The following code, from “lstdb.c”, demonstrates resizing the window in response to a column resizing. The actual resizing occurs on the **XIE_XVT_POST_EVENT** so that XI has completed its recalculation of the list size.

```
case XIE_COL_SIZE:
{
    LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

    list_info->list_resizing = TRUE;
    break;
}
case XIE_XVT_POST_EVENT:
    if ( xiev->v.xvte.type == E_MOUSE_UP )
    {
        LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

        if ( list_info->list_resizing )
        {
            RCT    rct;
            WINDOW win = xi_get_window( itf );

            xi_get_rect( itf, &rct );
            xvt_vobj_translate_points( win, xvt_vobj_get_parent( win ),
                                     (PNT*)&rct, 2 );
            xvt_vobj_move( win, &rct );
        }
    }
    break;
```

8.8 Using Groups

Groups are useful for two reasons. They give your application the opportunity to validate related information and to check database key references. For example, suppose that an application has two edit fields in a form that are related to one another such as an invoice and item number. In this application, it is important to verify that together the text a user enters in these edit fields reference a valid database record. You could also use a group if you had two columns in a list that were related such as minimum and maximum salaries. In this case, you would want to make sure that the minimum value is less than the maximum value, and notify the user that there is a problem if it is not.

You use groups when you want to validate a group of edit fields or cells when the user tries to move the focus out of the group. Inside the group, the user can freely edit text. It is not until he tries to move outside the group that the application is notified and can refuse to let the focus move outside the group.

8.8.1 Validating a Group of Edit Fields or Cells

When the user tries to move outside a group, your event handler will receive an **XIE_OFF_GROUP** event. Upon receiving this event, you can take one of two approaches to validate the contents of the group. The first approach is consistent with the recommendations we have made earlier in the *Using Forms* and *Using List* sections of this chapter. This approach involves looking at the data in the memory version of the record for the data to validate upon receiving the **XIE_OFF_GROUP** event. If for some reason the recommendations described earlier were not appropriate for your application, then you will not have the data stored in memory by the time you receive the **XIE_OFF_GROUP** event, and will need to take the second approach to validate the data in the group.

8.8.1.1 Validating a Group Using the First Approach

When validating a group of edit fields or cells using the first approach, recall that you have already validated text, converted the valid text to data, stored it in the memory copy of the record and marked it as “modified” by the time you receive an **XIE_OFF_GROUP** event. This is because you are always guaranteed to receive an **XIE_OFF_FIELD** or **XIE_OFF_CELL** events before an **XIE_OFF_GROUP** event.

Therefore, upon receiving the **XIE_OFF_GROUP**, you only need to look in the edit fields of that group in the memory record to see if together the data in the group makes sense. Since you have set up the group for a particular purpose, then you will already know what fields in the database you’ll need to look at, and you can do this without calling any XI functions.

The following example illustrates using the **XIE_OFF_GROUP** event to verify that the value of one edit field (minimum hours in this case) is less than the value of another edit field (maximum hours).

```
case XIE_OFF_GROUP:
    if ( form_info->changed && !emp_validate_hours( form_info->handle ) )
    {
        xvt_error( "Minimum hours cannot be greater than maximum." );
        xiev->refused = TRUE;
    }
    break;
```

The **emp_validate_hours** function can be found in “datdb.c” as follows.

```
BOOLEAN emp_validate_hours( long handle )
{
    EMPREC* rec = (EMPREC*)handle;

    return ( rec->minhrs <= rec->maxhrs );
}
```

8.8.1.2 Validating a Group Using the Second Approach

The second approach is more appropriate for you if you haven't responded to the **XIE_OFF_FIELD** or **XIE_OFF_CELL** events by validating the text and converting it to data. Therefore, by the time you receive the **XIE_OFF_GROUP** event, you will not have stored the data in the memory record. If this is the case, you will want to look at every cell or edit field in the group, get the text from each of them, convert the text to data and then validate the data as a whole.

To verify that the user entered valid text, you will need to take the group object handed to you in the **XI_OBJ** field of the XI event and call **xi_get_member_list** to get a list of edit fields or columns that belong to the group. If it is a group of columns, you would need to manufacture cells for them by using the **XI_MAKE_CELL** macro as explained in the *Using Cells* section of this chapter. After you have made the appropriate cell objects, you can pass their object pointers to **xi_get_text** to get the text for each of the cells. If it is a group of edit fields, you can just call **xi_get_text** since **xi_get_member_list** gave you the pointers you need.

8.9 Using Buttons

The only time an event is generated for a button is when the button is pressed by the user. Users can press buttons by clicking on them with the mouse or tabbing onto them and pressing the space bar or enter key. If you press the button with the mouse, it does not change the focus.

Buttons are especially useful to allow the user to perform some specific action. For example, your application might provide a form with an "Add Record", button to inform you that the user wants to write a record to the database. Whenever it receives an **XIE_BUTTON**, your event handler will typically switch on the control ID of the button and take whatever action is necessary. Since the only purpose of this event is to notify the application that something has happened, it is not logical to refuse this event.

In addition to using buttons to inform your application that the user wants to do something, you can also change their text and attributes. Button text is changed by calling **xi_set_text** with a pointer to the button object. Unless you are changing the text when responding to an **XIE_BUTTON**, you will not have an object pointer to the button to use to call **xi_set_text**, but you can get one by calling **xi_get_obj** with the control ID for the button.

8.9.1 Changing Button Attributes

You can change the behavior and appearance of a button by changing its attributes. The attributes are changed by calling **xi_set_attrib** with a pointer to the button object and passing it a bitwise OR'ed combination of values corresponding to the new attributes you want the button to have. The only attributes you can set for buttons are **XI_ATR_ENABLED** and **XI_ATR_VISIBLE**. If you want to disable or make invisible a button that has the focus, you will need to move the focus somewhere else on the interface before you can call **xi_set_attrib**. You move the focus by calling **xi_move_focus** with the appropriate XI object.

If you want to know whether or not a button is enabled or visible, you need to call **xi_get_attrib**. It returns the current attribute bit values for the button OR'ed together.

8.9.1.1 Disabling a Button

To enable or disable a button, you would call **xi_get_attrib** to get the attributes for the button, change the relevant bit and then call **xi_set_attrib** to give the button its new look. The following example illustrates disabling a button.

```
obj = xi_get_obj(itf, BUTTON1_CID);
xi_set_attrib(obj, xi_get_attrib(obj) & ~XI_ATR_ENABLED);
```

The following example illustrates enabling a button.

```
obj = xi_get_obj(itf, BUTTON1_CID);
xi_set_attrib(obj, xi_get_attrib(obj) | XI_ATR_ENABLED);
```

8.9.2 Checking Radio Buttons and Check Boxes

With XI, you can make buttons be either radio buttons, check boxes, or tab buttons. When you do this, it is necessary to check the button when the user presses the button. The function `xi_check` checks a radio button, check boxes, or tab button.

The following code, from “`lstlink.c`”, demonstrates using `xi_check`:

```
case SECTION_ONE_CID:
case SECTION_TWO_CID:
{
    SECTION_INFO* info = (SECTION_INFO*)xi_get_app_data( button );

    if ( form_info->cur_section != info )
    {
        change_section( form_info->cur_section, FALSE );
        change_section( info, TRUE );
        form_info->cur_section = info;
    }
    xi_check( button, TRUE );
    break;
}
```

If a radio button is part of a container, then calling `xi_check` unchecks any radio buttons that are also part of the container.

8.10 Using Static Text

Static text is used to put labels on the interface such as text to describe edit fields on a form. You may want to change the attributes of the text to make it visible or invisible, enabled or disabled, using `xi_set_attrib`. You may want to set the text of static text using `xi_set_text`. In either case, you will need to get a pointer to the appropriate static text object using one of the methods mentioned earlier in this chapter.

9

Managing Application Data

XI gives you the ability to associate a **long** integer with any XI object once the interface containing the object has been instantiated. The integer can be any kind of data you like, including a pointer cast into a **long**. The ability to associate a **long** integer with an XI object is especially useful in event driven programming, because you are often given an event that contains an object. Upon receiving the event, your application must somehow figure out what data is associated with the object.

To associate application data with an XI object, you call `xi_set_app_data`. To retrieve the data at a later time, you call `xi_get_app_data`. As you might expect, `xi_get_app_data` returns the **long** integer you set with `xi_set_app_data`. In the following example, you will see how to associate data with button objects. Later in this chapter you will see how associated data can be automatically freed when the object is freed by using XI's tree memory functions.

9.1 Associating Record Data with an Object

The “layered” form in the “Link List” example is a typical situation that uses `xi_set_app_data` and `xi_get_app_data` for buttons. The idea is that each of the buttons, which are tab buttons, have a set of XI object associated with them. These objects should be visible when the tab button is checked.

The following code, from “lstlink.c”, is the structure definition that will be used for application data for the buttons.

```
typedef struct
{
    int      count;
    XI_OBJ*  objs[ MAX_SECTION_OBJS ];
} SECTION_INFO;
```


The following code, from “lstlink.c”, demonstrates the initialization of these structures and the setting of application data for the buttons. This code appears in the **XIE_INIT** event for the window.

```
int          num;
SECTION_INFO* info[2];

info[0] = (SECTION_INFO*)xi_tree_malloc( sizeof( SECTION_INFO ), itf );
info[1] = (SECTION_INFO*)xi_tree_malloc( sizeof( SECTION_INFO ), itf );
xi_set_app_data( xi_get_obj( itf, SECTION_ONE_CID ),
                 PTR_LONG( info[0] ) );
xi_set_app_data( xi_get_obj( itf, SECTION_TWO_CID ),
                 PTR_LONG( info[1] ) );
form_info->cur_section = info[0];

for ( num = 0; textdefs[ num ].text != NULL; num++ )
{
    int section;

    section = textdefs[ num ].section - 1;
    if ( section != -1 )
        info[ section ]->objs[ info[ section ]->count++ ]
            = xi_get_obj( itf, TEXT_BASE_CID + num );
}

for ( num = 0; fielddefs[ num ].width != 0; num++ )
{
    int section;

    section = fielddefs[ num ].section - 1;
    if ( section != -1 )
        info[ section ]->objs[ info[ section ]->count++ ]
            = xi_get_obj( itf, FIELD_BASE_CID
                        + fielddefs[ num ].type );
}

```

The next step in this process is using the application data when a button is pressed. This is handled in the **XIE_BUTTON** event, which calls the **form_process_button** function. That code, from “lstlink.c”, is as follows:

```
void form_process_button( XI_OBJ* itf, XI_OBJ* button )
{
    FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

    switch ( button->cid )
    {
        ...
        case SECTION_ONE_CID:
        case SECTION_TWO_CID:
        {
            SECTION_INFO* info = (SECTION_INFO*)xi_get_app_data( button );

            if ( form_info->cur_section != info )
            {
                change_section( form_info->cur_section, FALSE );
                change_section( info, TRUE );
                form_info->cur_section = info;
            }
            xi_check( button, TRUE );
            break;
        }
        ...
    }
}

```

Notice how the return value of the call to `xi_get_app_data` is cast into the same structure that was set for the buttons. It then becomes quite easy to process the change in visibility for the layered objects.

If we were not using tree memory, it would also be necessary to respond to the `XIE_CLEANUP` event in order to free the allocated structures. Since they were parented to the interface object, they will be freed when the interface is deleted. Tree memory is covered more in the next section and the next chapter.

9.2 Using Tree Memory for Application Data

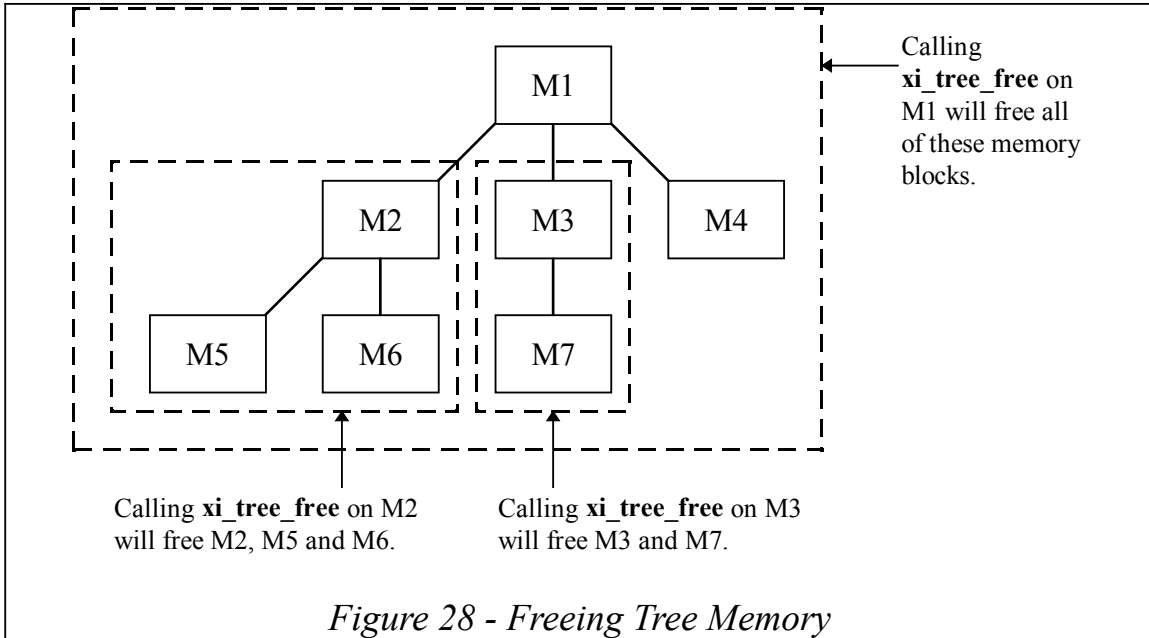
Your application can allocate memory for its data in such a way that it is freed automatically whenever an interface is closed or the XI object is deleted. XI has a mechanism called tree memory allocation that XI uses to allocate memory for objects in an interface hierarchy in such a way that when a parent object is freed, all of its children and grandchildren are automatically freed with it. Your application can take advantage of this by using XI tree memory functions to allocate the memory used for application data associated with XI objects. The result of using tree memory is that when the interface is deleted, all application memory associated with the interface is freed as well. If you want to use tree memory, your only constraint is that you must not try to reference any application data after your event handler has returned to XI after receiving an `XIE_CLEANUP` event. Details on using tree memory are discussed in the next chapter, *Memory Allocation*.

10

Memory Allocation

We highly recommend using tree memory allocation to increase the reliability of your program. Providence Software Solutions, Inc.. has implemented tree memory allocation, at first simply as a tool to make our own product more reliable. We did this because memory allocation bugs are one of the most annoying aspects of C programming since it is very easy to forget to free something, or free something when it shouldn't be freed. Problems with freeing memory always causes spurious bugs that are hard to track down and don't happen when you want to reproduce them. We made tree memory to eliminate a lot of this uncertainty.

Tree memory allocation is a means of keeping track of memory objects that are logical children of other memory objects. When you allocate a block of tree memory, you can associate it with a parent. When that parent is freed, all associated blocks are also freed.



The three main tree memory functions are `xi_tree_malloc`, `xi_tree_realloc`, and `xi_tree_free`. These are analogous to the normal memory allocation functions, but with a couple of differences. First, the function `xi_tree_malloc` takes a parent pointer. This sets up the association of parents and children. There is another function `xi_tree_reparent` that allows you to change this association at a later time. The other difference is that `xi_tree_free` will free everything that is associated with the block being freed.

Of course, all of these associations have to start somewhere, so you can pass `NULL` as the parent pointer to create a “root” for your allocation. These “root” blocks can be associated later using `xi_tree_reparent`. XI keeps track of these “root” blocks by making them children of an internal *root node* that is the parent of all “root” blocks.

NOTE: It is unnecessary and a bad idea to use tree memory allocation for C++ objects. The integrity of a C++ program depends on the execution of “destructor” functions for objects. The freeing of tree memory interferes with this constraint. However, the constructor and destructor mechanisms in C++ remove the need for tree memory.

10.1 Performance Considerations

Because of the way XI manages tree memory, it is better to have fewer children associated with each parent node because this will help the performance of the tree memory. As a general rule, freeing a node or reallocating a node is going to be as slow as the number of siblings that node has. For example, if you have one parent node with 200 children, a free or realloc operation is going to perform 10 times slower than during the same operation with a parent node of 20 children. This rule also applies to the root node that XI keeps internally for an object hierarchy.

Recall that whenever you allocate tree memory with a parent of `NULL`, this is equivalent to giving the root tree memory node one more child. Thus, if you allocate lots and lots of objects whose parents are `NULL`, then you will overload the root node with children, and will slow down the tree memory operations. By structuring your memory allocation trees properly you will avoid this problem. A properly structured tree is one where the average number of children per node is less than 100.

In addition, it is always faster to let `xi_tree_free` free all of the children of the nodes for you rather than trying to do it yourself. Whenever you delete a child from a tree, XI will rummage around in the link list and rearrange it properly. In contrast, when you ask XI to free an entire tree, XI knows to skip this step because the entire tree will be deleted.

Finally, each piece of memory allocated with **xi_tree_malloc** carries a 12 byte overhead. Therefore, if memory conservation is critical in your application, you may want to use another scheme.

10.2 Automatic Freeing of Tree Memory

Recall that when a piece of tree memory is freed, all of its children and grandchildren are freed with it. This is useful for having memory allocated for application data automatically freed when XI frees its memory.

The key to using tree memory in this fashion is knowing which types of XI objects use tree memory. Any object pointers in the interface hierarchy that are not pseudo-objects (cells and rows) are pointers to tree memory. If you use any of these pointers as the parent argument to **xi_tree_malloc**, then that memory that you allocate will be freed automatically whenever XI frees its memory. The interface object is most commonly used as a parent for your own allocations. Object definitions are also allocated using tree memory.

Now that you know which objects use tree memory, it is necessary to know when XI frees its memory. XI will free memory for an interface after the interface is deleted. The interface can be deleted in one of two ways: First, when the user clicks on a close box in the window, XI will send your event handler an **XIE_CLOSE** event. If the event is not refused, XI will close the window, send your event handler an **XIE_CLEANUP** event, and then free the XI object structures. The second way an interface can be deleted is when your event handler calls **xi_delete** with the interface object. In this case, your event handler will not receive an **XIE_CLOSE** event, but it will receive an **XIE_CLEANUP** event. It is not until after XI has sent your application an **XIE_CLEANUP** event that XI frees tree memory.

10.3 Debugging Tree Memory

A common problem with C programming is making sure that your application frees every piece of memory that it allocates. If an application does not free all the memory in a certain routine, as this routine gets called over and over, less and less memory is available to your application. This type of bug is called a memory leak.

The main advantage of using tree memory is that it helps you prevent memory leaks by allowing you to free an entire structure of memory with one call instead of using a separate call to free each node in the tree. In addition, there are times when you might allocate tree memory by calling **xi_tree_malloc** without calling **xi_tree_free** to free it later. This is the case if you are using **xi_tree_malloc** to allocate memory for data associated with an XI interface. Here, you will not need to explicitly call to **xi_tree_free** when responding to an **XIE_CLEANUP** event. The memory will be freed automatically.

If you are using tree memory for data other than that associated with an interface, and you suspect that you might have a memory leak, you can locate the problem by using the tree debugging features of the XI tool kit. When searching for the leak, you can start by calling **xi_tree_dbg** anywhere in your application. **xi_tree_dbg** will give you a status report of all of the memory blocks currently allocated.

One of the more useful ways to use tree debugging in this fashion is to set up operations in the program that shouldn't result in an accumulation of memory. For example, you could call **xi_tree_dbg** at the beginning of your program just to get a dump of all of the memory. Then, allocate memory by creating an interface, editing it, scrolling a list, etc. After you have done some of these operations, delete the interface, and call **xi_tree_dbg** again. At this point, the number of allocated items should equal to the number allocated at the start of the program, because theoretically you don't have more memory allocated after deleting the interface than before creating it. Thus, using tree debugging in this fashion will help you find some memory leaks without using the special debugging libraries.

The DEBUG file after calling **xi_tree_dbg** with no data allocated

```
TREE MEMORY DEBUG TRACE (Tree debug output)
=====
node 3b551858: par=00000000, sib=3b551858, ch=00000000
```

The DEBUG file after calling **xi_tree_dbg** with some data allocated

```

TREE MEMORY DEBUG TRACE (Tree debug output)
=====
node 3bd71858: par=00000000, sib=3bd71858, ch=32fb0000
  node 32fb0000: par=3bd71858, sib=330d0000, ch=00000000
  node 330d0000: par=3bd71858, sib=34ef0000, ch=34ff0000
    node 34ff0000: par=330d0000, sib=34ff0000, ch=00000000
  node 34ef0000: par=3bd71858, sib=32fb0000, ch=34f30000
    node 34f30000: par=34ef0000, sib=364f0000, ch=00000000
  ...
    node 33030000: par=35190000, sib=35210000, ch=00000000
  node 35110000: par=35030000, sib=35070000, ch=00000000

```

When you examined the debugging files illustrated, you might have noticed that these files only tell you what memory was allocated, not where. To find out where in the code the memory was allocated, we have provided another means to use XI tree debugging. By defining a preprocessor constant, **TREEDEBUG**, and linking a special XI library when compiling your application, XI can tell you the file and line number where each piece of memory was allocated. As you might suspect, using the alternate library makes your application less efficient, but you will have more information stored with each piece of memory allocated.

You can define the constant, **TREEDEBUGTREEDEBUG**, on the compiler command line when programming on all platforms except for the Mac. On the Mac, you will need to define it in a header file. Once you have defined the constant, you must recompile all of your *.c files so that the tree memory functions will be turned into macros. These macros will access the debugging version of XI internal functions with extra parameters. After recompiling and linking with the debugging version of the XI libraries, you run your application. However, when you look at the summary of memory usage, you'll see extra information attached to each memory item. If you come across a suspected leak, you can find out where the memory was allocated, and then deduce why it wasn't freed. An example file created by calling **xi_tree_dbg** is shown below.

The DEBUG file after calling xi_tree_dbg with some data allocated and TREEDEBUG defined

```

TREE MEMORY DEBUG TRACE (Tree debug output)
=====
node 1dd51888: par=00000000, sib=1dd51888, ch=1e250000, file=FIRSTNODE, line=0
  node 1e250000: par=1dd51888, sib=21d50000, ch=00000000, file=xi_test.c, line=568
  node 21d50000: par=1dd51888, sib=21ad0000, ch=21e50000, file=\xi\src\xi.c, line=362
    node 21e50000: par=21d50000, sib=21e50000, ch=00000000, file=\xi\src\xi.c, line=363
  node 21ad0000: par=1dd51888, sib=1e250000, ch=21bd0000, file=\xi\src\xi.c, line=1060
    node 21bd0000: par=21ad0000, sib=26750000, ch=00000000, file=\xi\src\xi.c,
line=1094
  node 26750000: par=21ad0000, sib=26550000, ch=26850000, file=\xi\src\xi.c,
line=1060
    node 26850000: par=26750000, sib=26cd0000, ch=00000000, file=\xi\src\xi.c,
line=1151
  node 26cd0000: par=26750000, sib=26bd0000, ch=26dd0000, file=\xi\src\xi.c,
line=1060

```

11

Modifying an XI Interface

When programming an application containing XI interfaces, you may need to add and delete XI objects after the interface has been instantiated. For example, you might have a spreadsheet where you “hide” and “show” columns. To “hide” the column, you simply delete the column object from the list. To “show” the column, you would define and instantiate the column object — in effect, “adding” it to the list. The “Memory” list is an example of this.

Changing the compositions of objects in an interface is what we mean by “modifying” it. While in previous chapters, you learned how to define and instantiate an entire interface tree, in this chapter you learn how to define and instantiate individual objects. In our discussion, we will show you how the convenience functions and `xi_create` operate internally so that you will have some insight into how these functions are used. In addition, you will learn how to adjust the size of the XVT window containing the interface so that it can grow or shrink to accommodate your modifications.

11.1 Adding Objects

When adding an object to an interface, you will need to do two things: define the object and instantiate it by calling `xi_create`. When defining it, you would most likely use the convenience function that creates the object definition structures for the object. To illustrate how to define and instantiate a single object, we have included two code examples in this section: adding a column to a list, and adding an edit field to a form.

11.2 Defining an Object

When reading the previous chapters, you saw that object definition structures are created for each object, and that those structures store information that XI needs to create the object. You saw that each definition contains two structures — an `XI_OBJ_DEF` which is generic to all objects, and a unique structure specific to the object such as an `XI_COLUMN_DEF` for a column or an `XI_FIELD_DEF` for an edit field. You also saw that convenience functions are used to create these structures and that they sometimes don’t set all of the fields used to describe an object. Because of this, you will need to know how to set any fields you want to use that aren’t normally set by the convenience functions.

Until now, you've seen how to call convenience functions in the context of defining an entire interface. When defining an object to be added to an existing interface, you will call the appropriate convenience as before, *except* that you pass in **NULL** for the parent. This is because the parent object has already been defined and instantiated, and you will not need to create a new definition for it. In fact, if you passed in a parent definition for the column, XI would try to attach the column to a different list than the one you have already created. Please remember that it is not until you call **xi_create** to create the column, that XI will know which list will get the column.

Of course, you can also add composite objects to an interface such as forms, lists and containers. To define a composite object, you would call the appropriate convenience function passing in **NULL** for its parent. Once you have an **XI_OBJ_DEF** for the composite object, you would add children definitions to it by calling the appropriate convenience function with the **XI_OBJ_DEF** for the composite object. When calling the convenience functions to define its children, you will need to pass in its object definition as the parent.

There is one other way to get an object definition and that is to get it from an existing object using **xi_get_def**. This is what happens in the "Memory" list example when you delete a column.

11.3 Instantiating an Object

Once you have created a stand-alone object definition, you are ready to call **xi_create** with the **XI_OBJ_DEF** for the object and a pointer to the object's future parent. To get the object pointer, you can call **xi_get_obj** with the interface object and control ID of the future parent.

The above discussion gives you a general idea of how to add an object to an XI interface. However, we didn't describe the details of adding an object to an interface in use. In the remaining portion of this section, we will describe the ways you will need to prepare the interface to receive an extra column or edit field. We will also give you samples of code to illustrate these two examples.

11.4 Adding a Column

Before you can add a column to a list, you will need to move the focus from the list to the interface. To move the focus, you can call either **xi_set_focus** or **xi_move_focus** with the interface object. **xi_move_focus** is a more gentle approach in that it will generate focus events so that you can validate a cell's data before moving the focus. In contrast, **xi_set_focus** will move the focus to another object without generating focus events.

In addition to moving the focus, you may need to resize the XVT window containing the interface to make room for the wider list. You will find a discussion on resizing the XVT window at the end of this chapter.

As mentioned before, the first thing you would need to do is define the column by calling **xi_add_column_def**, passing in **NULL** for the parent. Keep in mind that the *position* field in the **XI_COLUMN_DEF** structure determines where the column will be placed in the list. If you want to insert a column before another column of a certain position, set *position* to that column number. If you want to insert the column at the end of the list, then set *position* to a value greater than the number of columns.

Once you have an **XI_OBJ_DEF** for the column, the next thing you must do is get the object pointer of the column's future parent by calling **xi_get_obj** with the interface object and control ID for the list. Now you are ready to call **xi_create** to instantiate the column. XI will insert the column into the list and will generate cell request events for each cell in the column.

After **xi_create** returns, you may want to free the column definition by calling **xi_tree_free** or **xi_def_free** with the **XI_OBJ_DEF** to free both definition structures. You may also want to move the focus back to a cell in the list by calling either **xi_set_focus** or **xi_move_focus** with the appropriate cell pseudo-object, manufactured with the **XI_MAKE_CELL** macro or returned from **xi_get_focus**. The steps you'll need to take to add a column to a list is summarized in the following code example.


```

XI_OBJ_DEF *coldef;
XI_OBJ *list;

if ( itf && xi_move_focus( itf ) )
{
    coldef = xi_add_column_def( NULL, COL2_CID, XI_ATR_ENABLED, 0,
                               20 * XI_FU_MULTIPLE, 30, "Column 2" );
    coldef->v.column->position = 1;
    list = xi_get_obj( itf, LIST_CID );
    xi_create( list, coldef );
    xi_tree_free( coldef );
}

```

11.5 Adding an Edit Field

Adding an edit field to a form is a little simpler than adding a column to a list. In this case, you don't have to move the focus to the interface. Because edit fields are seldom created without a static text control to identify them, you might want to create both objects at the same time. However, you must do this with two separate calls to **xi_create** because **xi_create** can only define either trees of related objects or one object at a time. Since static text is a child of an interface and the edit field is a child of a form, the two objects are not related. Similarly, you could not add two edit fields at the same time because they are related to one another only by the parent they share, and the parent has already been instantiated.

To add an edit field to a form, you would need to define the edit field by calling **xi_add_field_def**, passing in **NULL** as the parent. Then, you would need to get the form object by calling **xi_get_obj**. Once you have the edit field definition and parent object, you can call **xi_create**. After the edit field is instantiated, you may want to call **xi_tree_free** or **xi_def_free** with the **XI_OBJ_DEF** to free the edit field definition. The following code illustrates how to add an edit field to a form.

```

XI_OBJ_DEF *fielddef;
XI_OBJ *form;

if ( itf && xi_move_focus( itf ) )
{
    form = xi_get_obj( itf, FORM_CID );
    fielddef = xi_add_field_def( NULL, FIELD2_CID,
                                13 * XI_FU_MULTIPLE, XI_FU_MULTIPLE,
                                16 * XI_FU_MULTIPLE, XI_ATR_ENABLED
                                | XI_ATR_BORDER | XI_ATR_VISIBLE,
                                FIELD2_CID, 20, COLOR_BLACK,
                                COLOR_WHITE,
                                COLOR_BLACK, COLOR_WHITE, COLOR_BLACK );
    xi_create( form, fielddef );
    xi_tree_free( fielddef );
}

```

11.6 Deleting Objects

In XI, there are two notions of “deleting an object”. When deleting a row in a list, XI will remove the contents of the row and automatically scroll the list to fill in the empty space. The list doesn't change. In contrast, when you delete a column, you are physically removing the column object from the interface and the list will become narrower (if the list is not a horizontally scrolling list). Therefore, the difference between deleting a row and deleting a column is that deleting a row does not change the size of the displayed list.

The function you call to delete an object is **xi_delete**. If you are deleting an object with children, the children will be deleted at the same time. Unlike when creating object definitions, you will not have to call **xi_tree_free** to free the objects. This happens automatically.

11.7 Deleting a Column

To delete a column, you call `xi_delete` passing it the object pointer to the column. To get the object pointer, you call `xi_get_obj` with the interface object and control ID for the column.

Because you cannot delete a column when any cell in the list has the focus, you will need to move the focus to the interface before you call `xi_delete`. You can move the focus by calling either `xi_set_focus` or `xi_move_focus`. The difference between the two functions is that `xi_move_focus` will generate focus events and `xi_set_focus` will not. After `xi_delete` returns, you may want to move the focus back to the list by calling either `xi_set_focus` or `xi_move_focus` with the appropriate cell pseudo-object, manufactured with the `XI_MAKE_CELL` macro.

Keep in mind that when you delete a column, XI will not generate any list events. This is because XI will not need to ask you to fill in any new information. The `XIE_COL_DELETE` event is only generated when the user drops a movable column off of the list in order to delete it. The following line of code demonstrates how to delete a column.

```
if ( itf && xi_move_focus( itf ) )
    xi_delete( xi_get_obj( itf, COL2_CID ) );
```

11.8 Deleting an Edit Field

To delete an edit field you call `xi_delete` with the edit field's object pointer. To get the object pointer, you call `xi_get_obj` with the interface object and control ID for the edit field. If the edit field you are deleting has the focus, you will need to move it to the interface before calling `xi_delete`. Depending on whether you want to receive focus events or not you can call either `xi_set_focus` or `xi_move_focus` to move the focus. After `xi_delete` returns, you may want to move the focus to another edit field in the form by either setting or moving it. The following lines of code demonstrate how to delete an edit field.

```
if ( itf && xi_move_focus( itf ) )
    xi_delete( field );
```

11.9 Resizing an XVT Window

After modifying an interface, you may want to resize the XVT window containing it. This is especially the case if you added an object that makes the interface larger than the client area of the XVT window containing it. To resize the window, you will need to do several things: get the XVT window for the interface; get the bounding rectangle for the modified interface; determine the correct client window size by adding white space to the bounding rectangle; position the window with respect to its parent window, and perform the window resizing operation.

From the user's perspective, the result of the above operations will be the following. First the user will see the new XI object displayed in the window, perhaps being cut off by the window if the modified interface is larger than the window. Then, the window will "grow" or "shrink" to accommodate the modified interface. If the window grows, the exposed portions of the interface will be redrawn.

Before you can resize the window, you will need to get the object pointer for the interface. Once you have the pointer, you can call `xi_get_window` to get the XVT window handle, and `xi_get_rect` to get the bounding rectangle for the modified interface.

After getting the new desired bounding rectangle of the window, it is necessary to translate the rectangle to the parent window's coordinate space. You do this by calling the XVT function `xvt_vobj_translate_points`.

The following code, from "lstdb.c", shows how to modify the XVT window size to match the current size of the interface.

```
RCT    rct;
WINDOW win = xi_get_window( itf );

xi_get_rect( itf, &rct );
xvt_vobj_translate_points( win, xvt_vobj_get_parent( win ),
                           (PNT*)&rct, 2 );
xvt_vobj_move( win, &rct );
```

12

Integrating XI with XVT

Applications

There are times when you will want to go outside of XI to do things in a window containing an XI interface. For example, you may want to draw graphical objects, use XVT controls, or provide menus. You can put XVT code in either your XVT event handler, or in your XI event handler. Putting XVT code in your XI event handler may make your application more modular.

In this chapter, we will describe how to use XVT drawing primitives to draw graphical objects, create and manipulate XVT controls, and put menus in the window with an XI interface. In all of these examples, we will be assuming that you will be putting the XVT code in your XI event handler.

12.1 Using XVT Controls

To create an XVT control, you will need to call the XVT function, **xvt_ctl_create**. The most appropriate time to call **xvt_ctl_create** is while processing an **XIE_INIT** event. Recall that an **XIE_INIT** event is the first event your event handler will receive after defining and instantiating your interface. This is also the appropriate time to allocate memory with **xi_tree_malloc**. We recommend that you use **xi_tree_malloc** because it will then be unnecessary to free the memory when the user closes the window. The memory will be freed automatically when the interface it is associated with is freed.

When responding to the **XIE_INIT** event, you will also need to associate data with the XI interface. You will need to call **xi_set_app_data** with the pointer to the interface object and a **long** for your application data. See *Managing Application Data* for more information on using XI functions to manipulate application data.

Once you have created the control, and have data to keep track of it, you can use XVT functions to manipulate the control in response to the events the control will generate. In addition, you will likely need to write code to update database records in response to the user manipulating the controls.

The other issue with XVT controls is that they are not a part of the standard XI keyboard navigation. If you want the tab key to work with the XVT controls, you will have to handle the **E_CHAR** event and determine the tabbing sequence for the XVT controls.

We have tried to provide all of the controls that you will need within the XI framework. We hope that you will not need to use the native XVT controls with the XI controls.

12.2 Drawing Graphics

Drawing graphics in a window with an XI interface is done with the XVT drawing functions. These functions are described in the *XVT Programmer's Manual*. Like other XVT code, you can use these functions in your XI event handler if you observe two rules. First, you should draw upon the **XIE_UPDATE** event. That event comes after XI is done drawing all of the rest of the interface. (However, if you are drawing things that you want to appear as “background” for the interface, you should draw during the **E_UPDATE** event in **XIE_XVT_EVENT**.) Second, you should always set the XVT drawing tools explicitly when you start to draw, including the clipping rectangle. In general, XI will ignore how you set drawing tools, and will change them to suit its needs. Therefore, you will need to explicitly set them every time your XI event handler receives an **XIE_UPDATE** event.

Since the XVT drawing functions require a **WINDOW** handle, you will need to call **xi_get_window** to get that handle.

12.3 Menus

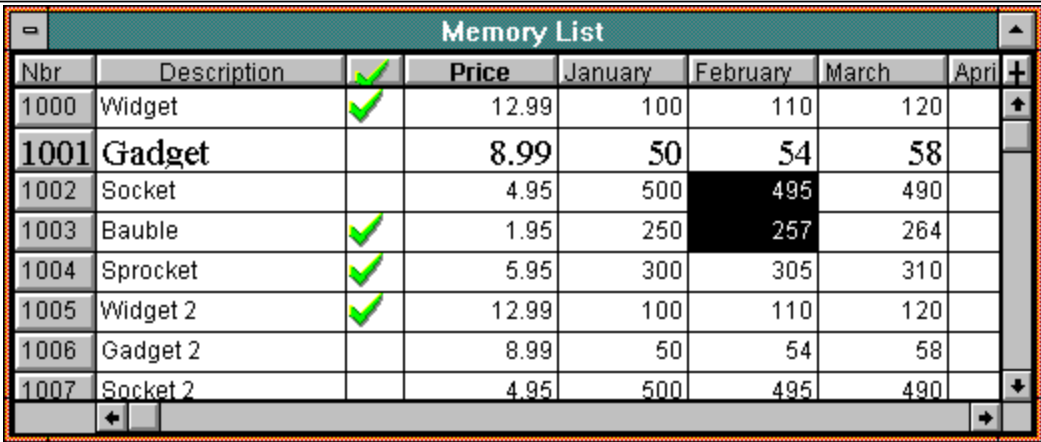
When adding menus to an XVT window containing an XI interface, you will need to create an XVT URL file as explained in the *XVT Programmer's Manual*, or use the XVT function **xvt_menu_set_tree**.

You will also need to create a switch statement to respond to the **XIE_COMMAND** event. Notice that in the case of menus, your event handler should respond to the XI event instead of the corresponding XVT event coming in through **XIE_XVT_EVENT**. To understand how the event is generated, whenever a user selects a menu item, XVT generates an **E_COMMAND** event. Upon receiving the **E_COMMAND** event, XI will stuff it into **XI_EVENT**; thereby turning it into an **XIE_COMMAND** event. When responding to the **XIE_COMMAND** event, you determine which menu item was selected by switching on the XVT menu tag (defined in the URL file or in the XVT **MENU_ITEM** structure).

Appendix A

The XI Example

This appendix is simply a catalog of the various screens that appear in the example program. It's main purpose is to let you know what you should be seeing when you run the examples. All of the screens were captured on MS-Windows. The exact appearance will vary on other platforms, with character-based being radically different.



Memory List							
Nbr	Description	✓	Price	January	February	March	April
1000	Widget	✓	12.99	100	110	120	
1001	Gadget		8.99	50	54	58	
1002	Socket		4.95	500	495	490	
1003	Bauble	✓	1.95	250	257	264	
1004	Sprocket	✓	5.95	300	305	310	
1005	Widget 2	✓	12.99	100	110	120	
1006	Gadget 2		8.99	50	54	58	
1007	Socket 2		4.95	500	495	490	

Figure 29 - The Memory List

The memory list demonstrates colors, fonts and attributes in cells, icons in cells, variable row heights. It also allows for user resizing, deleting and adding of columns. It uses an in-memory array to keep track of the data for the list.

Linked List			
Add All Recs		Add One Rec	Delete All Recs
Delete Current Rec		Delete Selected Recs	
Number	Date	Description	Who
1	01/01/95	This is description #1 which is long enough for word wrap.	Person 1
2	01/01/95	Alternate description #1	Person 2
3	01/01/95	Miscellaneous description #1	Person 3
4	01/01/95	Description #2	Person 4

Figure 30 - The Linked List

The linked list uses a doubly linked list as its data structure for the list data. It demonstrates gridded buttons, cells with word wrap, drop down lists from cells and row insert and delete. Double-clicking on a cell will popup an editing dialog as show below.

Edit Item	
Record #	1
<div style="display: flex; justify-content: space-between;"> {Descr} Other </div>	
Date	01/01/95
Who	Person 1 ▼
Estimated Hours	2
Actual Hours	0
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

Figure 31 - The Linked List Change Dialog

The linked list dialog is a modal XI window. It also demonstrates the “layering” of XI controls within the interface. The “Who” edit field demonstrates a drop down list on an edit field.

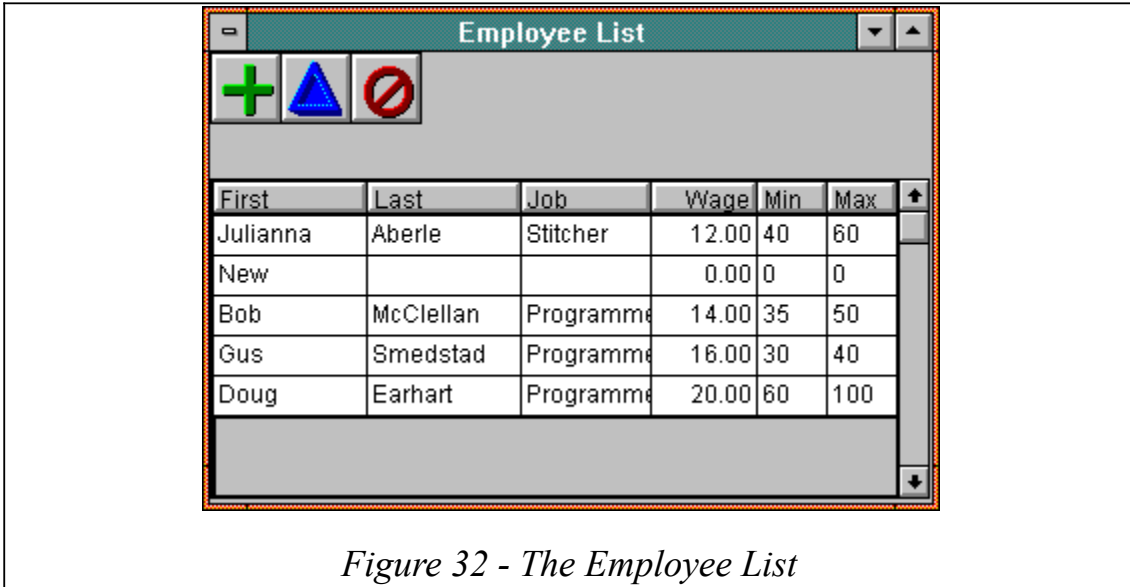


Figure 32 - The Employee List

The employee list uses a text file database as its data structure. It is the list that will probably be most interesting to you if you are using a database system. It also demonstrates icon buttons and grouped columns to validate the minimum and maximum hours.

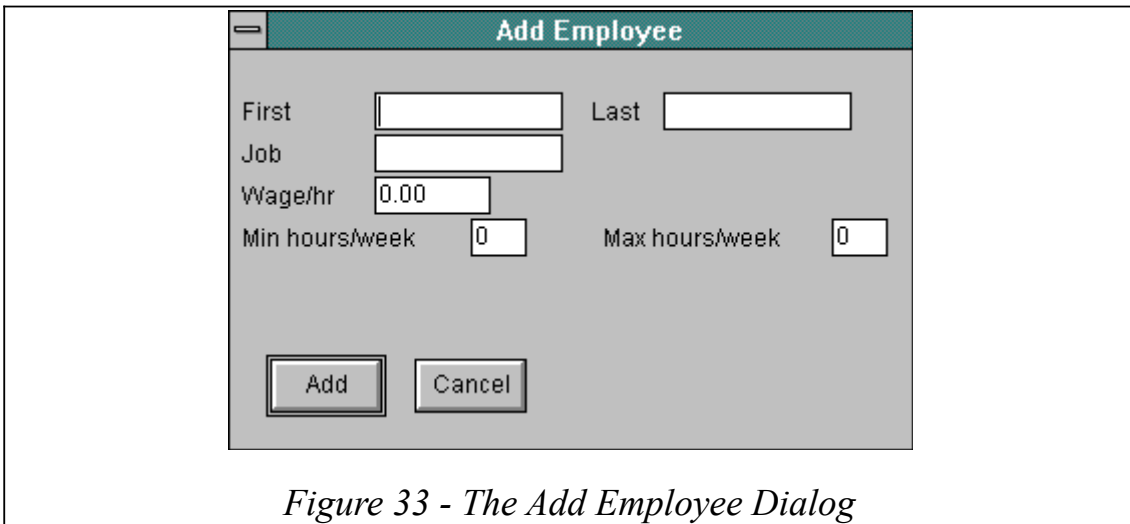


Figure 33 - The Add Employee Dialog

The add employee dialog is used to add employee records to the database. It demonstrates updating a list from a dialog and grouped edit fields to validate the minimum and maximum hours.

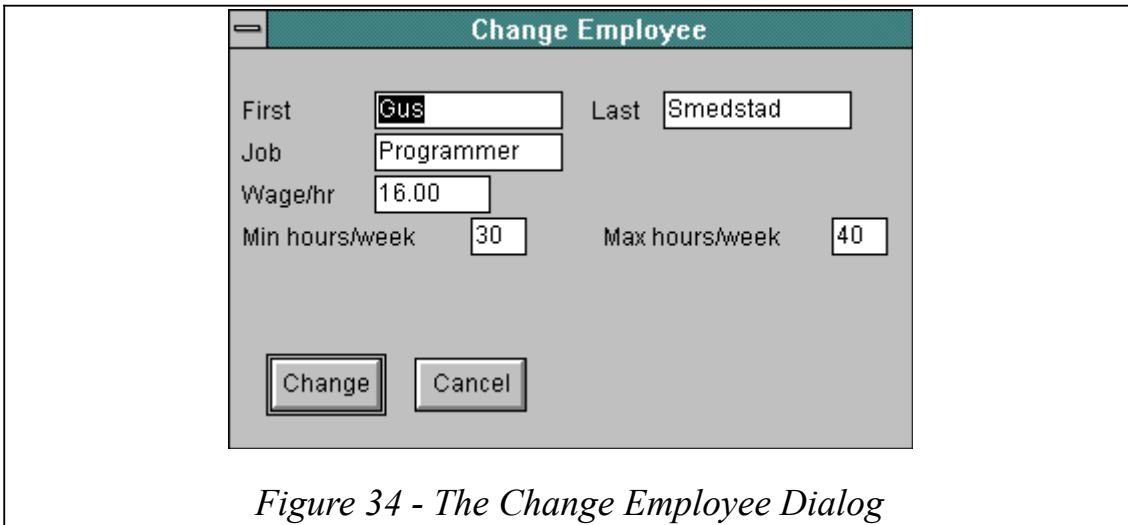


Figure 34 - The Change Employee Dialog

The change employee dialog is very similar to the add employee dialog. It also demonstrates updating a list from a dialog.

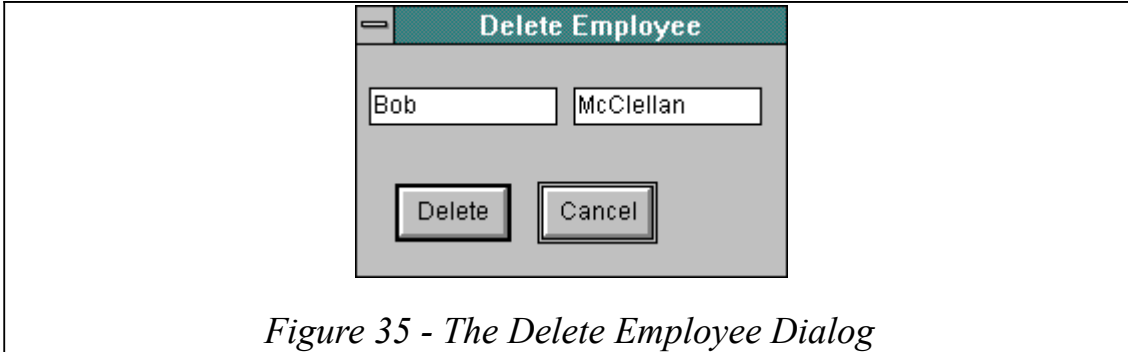


Figure 35 - The Delete Employee Dialog

The delete employee dialog demonstrates updating a list from a dialog and disabled edit fields.

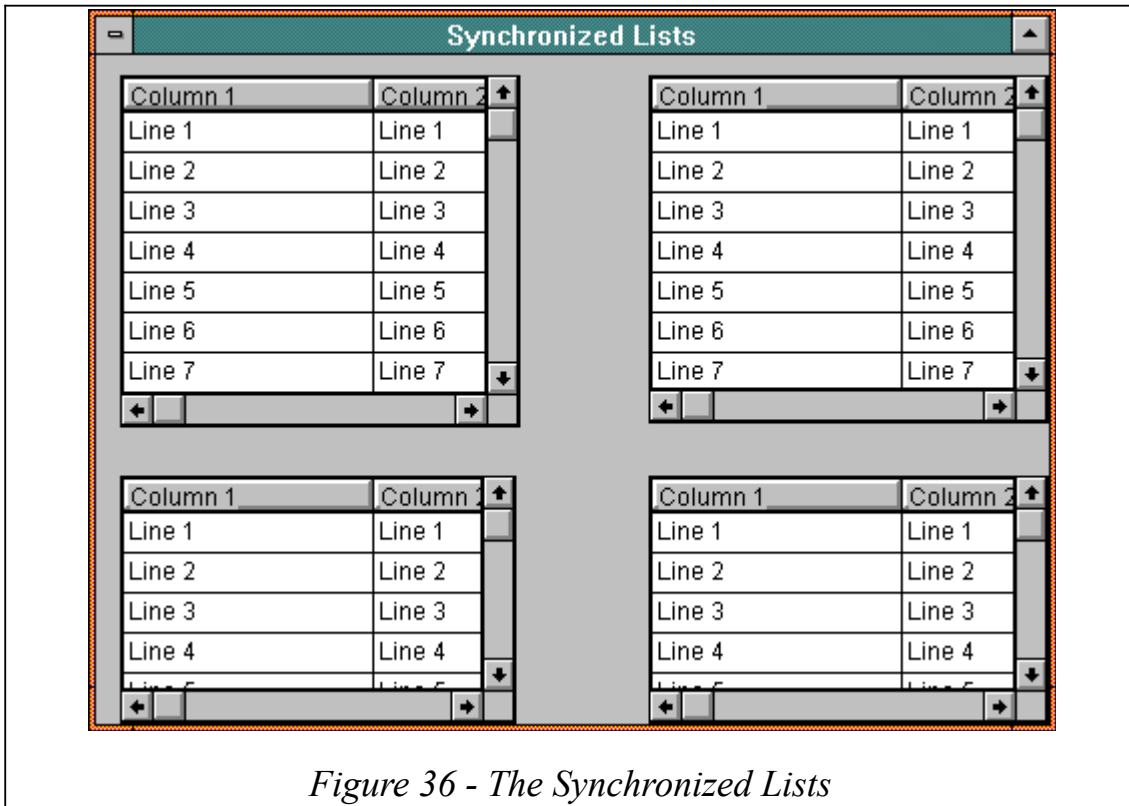


Figure 36 - The Synchronized Lists

The synchronized lists interface demonstrates lists that are connected so that they scroll together. It also demonstrates manual resizing of lists.

Index

A

- attribute, changing • 88
- attributes, fields • 88
- automatic_back_color • 21

B

- button attributes • 117
- buttons • 117

C

- cell object, making • 108
- cell request event • 98, 107
- cells, using • 107
- character filtering • 109
- children of objects • 84
- column attributes • 114

- column events • 115
- column heading, changing • 114
- column object, getting from a cell • 114
- column width, changing • 114
- column, adding • 127
- column, using • 113
- columns, deleting • 129
- convenience functions • 12
- creating an interface • 63

D

- drawing in an interface • 132
- drop_and_delete • 27

E

- edit fields • 86
- edit_menu • 22
- event handler • 68
- event handler per object • 11
- EVENT_HANDLER • 66
- events • 9
- events, scrolling • 103

F

- field attributes • 88
- field, adding • 128
- field, deleting • 129
- filtering characters • 86
- focus model • 73
- focus movement • 99
- form events • 79
- forms, using • 89
- forms, validating contents • 89

G

- getting an object from an event • 83
- getting an object pointer • 83
- group, using • 116
- group, validating • 116

I

- interface events • 76
- interface, modifying • 126

L

- list attributes • 105
- list events • 76
- list size, changing on E_SIZE • 106
- list, scroll bar • 104
- lists, using • 93

M

- main • 1, 4, 64
- menu_bar_rid • 22
- menus • 132
- metatab • 32

N

- non-refusable events • 73

P

- parent of objects • 85

R

- radio button, checking • 118

- record request events • 111
- records, managing • 94
- refusing events • 71
- retrieving records • 97
- row object, making • 111
- row, deleting • 112
- rows, using • 110

S

- scrolling a list • 102
- scrolling the list to a record • 104
- static text, using • 118

T

- TREDEDEBUG • 125

V

- validating fields • 87
- virtual interface • 73

X

- XI Interface • 7

- xi_add_button_def • 48, 50, 60
- xi_add_column_def • 32, 59
- xi_add_container_def • 14, 60
- xi_add_field_def • 42, 43, 44, 58, 93
- xi_add_form_def • 14, 57
- xi_add_group_def • 14, 61
- xi_add_line_def • 14, 54, 62
- xi_add_list_def • 14, 59
- xi_add_rect_def • 14, 61
- xi_add_text_def • 14, 61
- XI_ATR_AUTOSCROLL • 32, 43, 58, 59
- XI_ATR_AUTOSELECT • 32, 42
- XI_ATR_ENABLED • 25
- XI_ATR_NAVIGATE • 30
- XI_ATR_PASSWORD • 33, 43
- XI_ATR_READONLY • 32, 42
- XI_ATR_RJUST • 32, 43, 53
- XI_BTN_TYPE • 47
- xi_cell_request • 31, 108
- xi_check • 47, 50, 118
- xi_create • 12, 13, 19, 20, 54, 62, 63, 64, 84
- xi_create_itf_def • 14, 56
- xi_delete • 129
- xi_delete_row • 112
- xi_draw_line • 52
- xi_event • 68
- XI_EVENT • 69
- XI_EVENT_HANDLER • 56
- xi_get_app_data • 119
- xi_get_attrib • 19, 88
- xi_get_def • 27
- xi_get_def_rect • 63
- xi_get_list_info • 100, 110
- xi_get_member_list • 84, 89
- xi_get_obj • 84, 89
- xi_get_rect • 129
- xi_get_text • 83, 87
- xi_get_window • 129
- xi_init • 4, 65
- xi_insert_row • 112
- XI_INTERNAL • 20, 92
- XI_ITF_DEF • 14, 63

XI_LIST_DEF • 14
 XI_MAKE_CELL • 85, 107, 108
 XI_MAKE_ROW • 85, 111
 xi_move_column • 27
 xi_move_focus • 88
 XI_OBJ_DEF • 14
 xi_obj->v.field->tab_cid • 92
 XI_PREF_3D_LOOK • 53
 XI_PREF_COLOR_CTRL • 53
 XI_PREF_COLOR_DARK • 53
 XI_PREF_COLOR_LIGHT • 53
 XI_PREF_NATIVE_CTRL • 48, 50, 52
 xi_scroll • 31, 103, 104
 XI_SCROLL_FIRST • 31
 xi_scroll_rec • 31
 xi_set_app_data • 119
 xi_set_attr • 19, 88, 105
 xi_set_bufsize • 43
 xi_set_fixed_columns • 30
 xi_set_focus • 88
 xi_set_sel • 87, 101
 xi_set_text • 9, 86, 101
 xi_tree_dbg • 124
 xi_tree_free • 62, 64, 123
 xi_tree_malloc • 123
 xi_tree_realloc • 123
 xi.h • 4
 XIBT_BUTTON • 47
 XIBT_BUTTON_CHECKBOX • 48
 XIBT_BUTTON_RADIOBTN • 48
 XIBT_CHECKBOX • 47, 50
 XIBT_RADIOBTN • 47, 50
 XIBT_TABBTN • 47
 XIE_BUTTON • 28, 52, 79, 92
 XIE_CELL_REQUEST • 10, 36, 76, 77, 97, 103, 107
 XIE_CHAR_CELL • 77, 109
 XIE_CHAR_FIELD • 86
 XIE_CHG_CELL • 77
 XIE_CHG_FIELD • 86
 XIE_CLEANUP • 71
 XIE_COL_DELETE • 77, 115
 XIE_COL_MOVE • 77, 115
 XIE_COL_SIZE • 77, 115
 XIE_COMMAND • 80
 XIE_DBL_CELL • 77
 XIE_GET_* • 97
 XIE_GET_FIRST • 36, 76, 96
 XIE_GET_LAST • 36, 76, 96
 XIE_GET_NEXT • 36, 76, 96, 103
 XIE_GET_PREV • 36, 76, 96
 XIE_INIT • 50
 XIE_OFF_* • 97
 XIE_OFF_CELL • 10, 77, 109
 XIE_OFF_COLUMN • 77
 XIE_OFF_FIELD • 83, 86, 87, 92
 XIE_OFF_GROUP • 77
 XIE_OFF_LIST • 77
 XIE_OFF_ROW • 77, 102
 XIE_ON_* • 97
 XIE_ON_CELL • 25, 32, 77
 XIE_ON_COLUMN • 32, 77
 XIE_ON_FIELD • 42

XIE_ON_GROUP • 77
XIE_ON_LIST • 25, 77
XIE_ON_ROW • 25, 77
XIE_REC_ALLOCATE • 76, 94, 95, 103
XIE_REC_FREE • 76, 97
XIE_ROW_SIZE • 77
XIE_SELECT • 77
XIE_UPDATE • 80
XIE_VIR_PAN • 80
XIE_XVT_EVENT • 22, 80
XIE_XVT_POST_EVENT • 22, 80
XIT_COLUMN • 31
XIT_FORM • 8
XIT_GROUP • 8
XIT_ITF • 19
XIT_LINE • 8
XIT_LIST • 8, 24
XIT_RECT • 8
xvt.h • 4
XVT/CH • 48